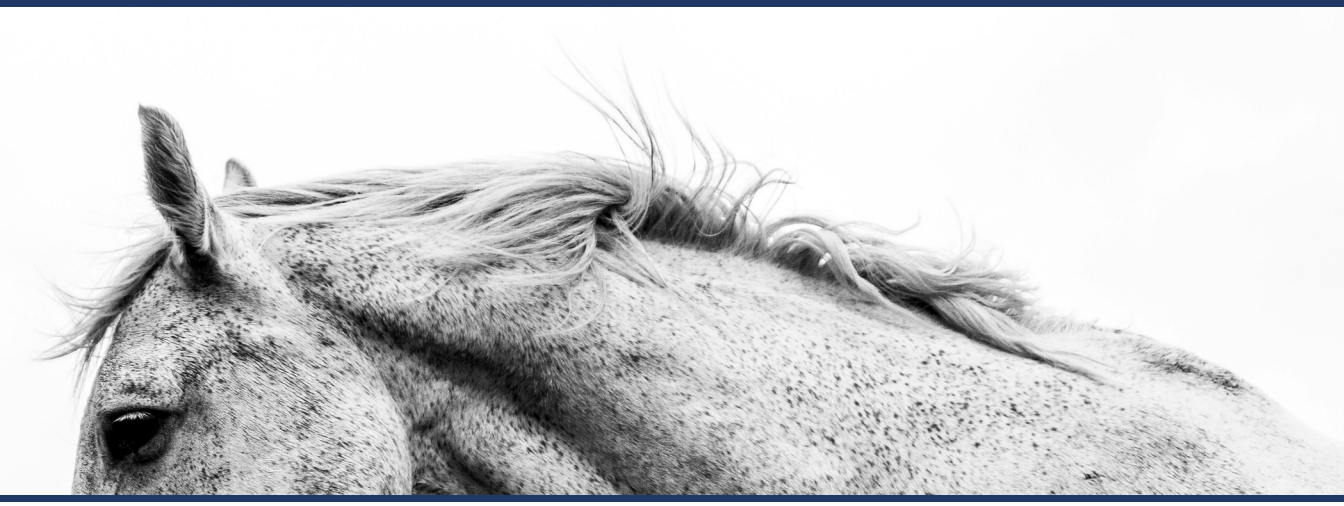
CYBERPSYCHOTIC MONOGRAPH SERIES IN APPLIED PROGRAM ANALYSIS · NO. 4



CYBERPSYCHOTIC VULNERABILITY RESEARCH

Lectures on Memory Safety, Fuzzing, and the Discipline of Finding Bugs

JACOB A.

THIRD EDITION

CYBERPSYCHOTIC PRESS

CYBERPSYCHOTIC VULNERABILITY RESEARCH

Lectures on Memory Safety, Fuzzing, and the Discipline of Finding Bugs

Jacob A.

Reader in Program Assurance, Third Bench
Institute for the Study of Adversarial Computation
Ninth Street Campus

THIRD EDITION, CORRECTED PRINTING

Cyberpsychotic Press · Chicago, Illinois

APPROVED SYLLABUS TEXT · SYSTEMS DIVISION ASSURANCE CANDIDATES

COMPACT RESTRICTED · NOT FOR UNSCOPED DISTRIBUTION

CYBERPSYCHOTIC PRESS

The Institute for the Study of Adversarial Computation

Founded 1987 as *The Cyberpsychotic Quarterly*, a stapled newsletter of nine pages distributed at cost; incorporated as a press in 1996 upon the publication of T. Vance's *Fault Injection*.

First edition 2011. Second edition 2017. Third edition 2024; corrected printings 2025, 2026.

Reviewed and cleared for instructional use by the ARGUS review board, Directorate of Assurance, under teaching exemption T-22.

Distribution restricted to credentialed personnel of the Assurance Compact, its sixty-three signatory laboratories, and matriculated candidates of the Institute. Every reader of this book becomes, on finishing it, part of somebody's threat model.

This book is an attack surface. Please treat it as one.

Set in Palatino by the author, who has been asked to stop doing his own typesetting and has declined.

Cover photograph by Avi Theret, made in Wales and published 3 October 2020; used under the Unsplash License, which permits commercial use without attribution. It is credited here anyway. Converted to greyscale and cropped for this edition; not otherwise altered.

Catalogue: CPP-MS-APA-4 · Syllabus codes: VR-201, VR-202, VR-311.

For the autumn 2011 cohort —
all fourteen of you, and for every maintainer who answered.

And for Ampersand, who is not a source of entropy,
whatever §B.3 says.

Preface

The first bug I ever found, I found by accident, and I have spent the rest of my working life trying to make the accident repeatable.

It was 1994 and I was supposed to be writing a mail filter. I typed an absurdly long subject line into my own test harness (long because I had fallen asleep on the keyboard, which is the calibre of experimental method I brought to the work at that age) and the program printed about forty bytes of somebody else's memory at me and then carried on as though nothing had happened. I did not understand what I had seen. I spent two days on it. At the end of the two days I still did not understand what I had seen, but I had noticed the thing that mattered, which was that the program had a second behaviour underneath the documented one, and that nobody had put it there on purpose.

That noticing is the talent, and I have never worked out how to teach it. Fortunately I have never had to. Everybody who ends up in this room already has it; you would not have applied otherwise. What I can teach is everything that comes afterwards: how to go looking on purpose instead of by accident, how to tell when you have actually found something, and how to convince a stranger who is busy and sceptical and has forty other things in their inbox.

Let me say plainly what kind of book it is. There are two kinds, this is the second, and the difference matters more in this subject than in most. The first kind promises power. It has a hooded figure on the cover, or a green terminal, and the chapters are named after verbs: *own*, *pivot*, *escalate*. It gives you nine tools and no model of anything, so that when the tenth situation shows up you have nothing to think with. The tenth situation always shows up. The second kind is a textbook: it has problem sets, it assumes you can read C and hold a pointer diagram in your head, and it will not tell you that you are special, because by the time you can read it, finding bugs is your job description, and nobody's job description is interesting.

The subject is vulnerability research: finding the places where software is wrong in ways that matter, and proving it well enough that somebody fixes it. I have spent my career on the finding rather than the using, and I will give the reason once and then drop it. The Compact will teach you plenty of ways to turn a defect into an outcome; it pays me to teach the one part of the work whose entire output is a patch. I am aware that this sounds like a moral position and I would rather you took it as a practical one. The

field is small. Your name outlives your access to any particular system, and the people still doing this at fifty are, almost without exception, the ones who can list the bugs they closed rather than the boxes they opened. I would like you to still be doing this at fifty.

Prerequisites, briefly. C to the point where pointer arithmetic does not frighten you. A working picture of the stack and the heap. Enough x86-64 or AArch64 to read a function slowly with a reference open. Slowly is fine; nobody reads assembly quickly, and the people who claim to are reading the parts they have seen before. A shell you are at home in, and a debugger you have used in anger rather than in a tutorial. Appendix A is a refresher for those of you who last wrote C some years and two frameworks ago; there is no shame in it and I have reread my own copy.

What you do not need is prior offensive practice. Some of you have it. If you arrive with a capture-the-flag habit, good; if you arrive with a directory of scripts whose provenance you would rather not put on a form, I would like you to set it down at the door for one term. Underground practice is a tangle of true things, superstition, and the occasional felony, and until you can compute which is which the tangle is what will hurt you.

On voice: I lecture in the first person and I have not bothered to launder it out for print. The footnotes are where I put things that are true but not on the syllabus.[1] Where something is dangerous, whether to a machine, to a person, or to your legal standing, it goes in a marked box, and those boxes are the only part of this book I will ask you to memorise.

There are things this book does not contain, and I would rather you learned about them here than on page 200.

There are no weaponised exploits in this book. We go as far as the reproducer and the impact argument, and then we stop. You will learn to build a minimal crashing input, to find the line that is actually wrong, to say honestly what an attacker could and could not do with it, and to hand the whole package to somebody who can fix it. Turning a memory-safety defect into reliable code execution against a modern target is a real discipline with real practitioners and it is not this course. I will add, because candidates always ask and are always disappointed: in twenty-odd years of filing reports I have needed a working exploit to get a bug taken seriously perhaps four times, and on two of those occasions I was wrong and a better write-up would have done it.

Nor is there a target list. Everything here is demonstrated either on toy code I wrote to be broken or on public vulnerabilities that somebody else found, disclosed, and got

[1] The theory group considers this unserious. I have read a great many incident reports and I can report that nobody has ever been injured by a footnote, whereas a large number of people have been injured by documentation too dignified to say *this is the part everyone gets wrong*.

fixed, and whose finders have published better accounts than my summaries. The exercises are scoped to code you own or code you have written permission to touch. That constraint is not squeamishness; it is the line between a profession and a hobby with defendants.

Thanks are owed. To the Institute, for tolerating a third edition. To fourteen cohorts of candidates, whose wrong answers told me where the explanations were broken. If a section of this book is unusually clear it is because a candidate was unusually confused at me about it. To my late colleague Yelena Sorokin, who found the topology underneath the tooling and argued with me about Chapter 5 until a month before she died, and who was right about most of it. And to the maintainers of the projects I have reported to over the years, most of whom were a good deal kinder to me than the situation strictly required.

J. A.
Ninth Street
Michaelmas, 2024

Notation and Conventions

Program text, file names, tool names, and identifiers appear in **bold** when they occur inside a sentence, and in a typewriter face only in displayed listings. I set it up that way after years of watching people's eyes stumble every time the type changed mid-line. A shell prompt of **$** means an unprivileged user; **#** means root, and every **#** in this book is a decision somebody signed for. Sizes are bytes unless marked. Addresses are hexadecimal and, where they appear, fabricated. The symbol □ ends a proof; the symbol ▲ ends a worked example; nothing ends a war story.

CVE identifiers are given in full on first use in a chapter and are real: every public vulnerability discussed in this book was found by someone else, disclosed to a vendor, and fixed, and the write-ups by the people who found them are better than my summaries. Read them. Institutional names, cohort numbers, internal incident references, and all named colleagues are invented; see the colophon.

Symbol	Name	First appears
$\mathscr{S}$	attack surface	§1.1
$\mathscr{T}$	trust boundary	§1.1
$\mathscr{I}$	input space of a harness or interface	§1.2
$\rho(v)$	reachability of state v	§1.2
$\chi(v)$	attacker control over v	§1.2
κ	consequence class	§1.2
Σ	sanitiser (an oracle on program states)	§2.5
$C(Q)$	edge coverage achieved by corpus Q	§4.2
Q	corpus (the fuzzer's working set)	§4.2
r	throughput, executions per second	§4.2
d	constraint depth of a bug	§4.2
p	per-execution discovery probability	§4.2
N	executions expended	§4.2
τ	expected time to first trigger	§4.3
$\mathscr{H}$	harness	§5.3
ϕ	harness fidelity	§5.3
$L(s,k)$	feasible-path linkage, source to sink	§5.5
Y	yield of an engagement	§6.2
m	match ratio, researcher to target	§6.2
F	fatigue integral	§6.4
$H(\Delta)$	sustainable effort over duration Δ	§6.4
δ	a minimisation step (*ddmin*)	§8.2

Three questions are asked so constantly in this book that they carry small capitals throughout: REACHABILITY (can the code be made to run?), CONTROL (how much of the resulting state does the attacker choose?), and CONSEQUENCE (what can that state be made to do?). When I write "Miller" with no citation I mean Barton Miller's 1990 report on random testing of Unix utilities, the way a physicist writing "Newton" means the *Principia*. When I write "the Handbook" I mean Dowd, McDonald, and Schuh, *The Art of Software Security Assessment* (2006), which is longer than this book, older than this book, and on the subject of manual auditing, better than this book.

What in This Book Is Invented

This book is fiction in its framing and instruction in its content. You are entitled to know which is which before you start rather than after you finish, so here is the list. The colophon repeats it at the back with more detail on the sources.

Invented. Cyberpsychotic Press. The Institute for the Study of Adversarial Computation, its Ninth Street campus, the Assurance Compact, the Bench grades, the ARGUS review board, the syllabus codes, and every cohort and incident reference. Every named person: Jacob A., T. Vance, Y. Sorokin, D. Prather, R. Okoye, L. Wu, and the works and theorems attributed to them. The archive proportions in §3.2. The doctrine values in the tables of Chapter 7, which are the author's working defaults and were described in earlier printings as composites over engagements that did not happen.

Modelled rather than measured. (6.1), the yield relation, whose functional form is borrowed from electrical engineering and which nobody has fitted to data about this profession. (6.2), the fatigue equation, likewise. Both carry a note at the point of use, because a reader who meets an equation should not have to remember a page from the front matter to know what it is.

Real, and checkable. Everything else. The tools and their documented mechanisms and costs; the vulnerabilities discussed by identifier and their root causes as published by their finders; the history; the published proportions and figures; the disclosure policies; and the literature cited in the Notes and Sources sections. Those are written out of universe and mean exactly what they say. Where this book paraphrases a source it may have paraphrased it badly, and the source is better than the paraphrase.

Arguments that carried invented attributions in earlier printings have had them withdrawn: the completeness argument of §3.2, which is ordinary path induction and needed no name on it, and the concavity claim of §6.4, which now cites the working-hours and sleep-restriction literature. The characters remain. They are no longer sources.

Contents

Preface **ix**

Notation and Conventions **xiii**

1 The Shape of the Problem **1**
1.1 What a vulnerability is . . . 1
1.2 Three questions, restated . . . 3
1.3 A short and partial history . . . 5
1.4 Why the method is not optional . . . 6
Notes and Sources . . . 7
Exercises . . . 8

2 The Machine Underneath **9**
2.1 What we model, and what we do not . . . 9
2.2 Objects, lifetimes, and the two safeties . . . 9
2.3 Undefined behaviour, and the term that changes everything . . . 10
2.4 Where the objects live . . . 11
2.5 Instruments . . . 12
Notes and Sources . . . 15
Exercises . . . 15

3 Statics: Reading Code for Bugs **17**
3.1 The audit problem . . . 17
3.2 Sources, sinks, and the completeness argument . . . 17
3.3 The arithmetic that lies . . . 19
3.4 Two case studies in the sixth shape . . . 22
3.5 Reading with a threat model . . . 23
3.6 Variant analysis, and how one bug becomes eleven . . . 24
Notes and Sources . . . 25
Exercises . . . 25

4 Dynamics: Fuzzing **27**
4.1 From reading to running . . . 27
4.2 The arithmetic of the constraint chain . . . 27
4.3 How long should this take? . . . 28
4.4 Making the mutations count . . . 29
4.5 Corpus and sanitiser hygiene . . . 29
4.6 The plateau, and what to do about it . . . 30
Notes and Sources . . . 31
Exercises . . . 32

5 Topology: Coverage, Harnesses, and Reach **33**
5.1 The coverage theorem, and what you cannot find . . . 33
5.2 What kind of surface is your target? . . . 34
5.3 Doors: the harness, and why it is the whole job . . . 35
5.4 When the fuzzed volume is correctness . . . 37
5.5 Links: dataflow, taint, and feasible paths . . . 37
Notes and Sources . . . 38
Exercises . . . 39

6 The Researcher: Yield, Match, and Limit **40**
6.1 The researcher as a component . . . 40
6.2 Effort, match, and the yield theorem . . . 40
6.3 Redline: the cost of total coverage . . . 41
6.4 Fatigue, and the sustainable-effort bound . . . 42
6.5 What “talent” turns out to be . . . 43
6.6 Substitutions: pipelines, analysers, and agents . . . 43
Notes and Sources . . . 44
Exercises . . . 44

7 Five Standard Engagements **46**
7.1 How to read a pattern . . . 46
7.2 Pattern I: the drive-by . . . 47
7.3 Pattern II: the harness campaign . . . 47
7.4 Pattern III: the deep audit . . . 48
7.5 Pattern IV: the moving target . . . 49
7.6 Pattern V: the live target . . . 50
Notes and Sources . . . 51

Exercises . . . 52

8 The Proof of Concept **53**
8.1 What a proof of concept is for . . . 53
8.2 Minimisation . . . 53
8.3 Root cause: the crash site is not the bug site . . . 54
8.4 The artefact . . . 55
8.5 The impact argument, and its honest limits . . . 55
8.6 The report . . . 56
8.7 Disclosure . . . 57
Notes and Sources . . . 58
Exercises . . . 58

9 What We Do Not Know **60**
9.1 Why this chapter exists . . . 60
9.2 Problem 1: the harness gap . . . 60
9.3 Problem 2: residual defect estimation . . . 61
9.4 Problem 3: exploitability . . . 61
9.5 Problem 4: crashes to bugs . . . 61
9.6 Problem 5: what the agents do . . . 62
Notes and Sources . . . 62
9.7 Commencement . . . 63

A Toolchain Refresher **65**
A.1 Five facts about memory . . . 65
A.2 The debugger in nine commands . . . 65
A.3 The flags that matter . . . 66
A.4 Reading a stack trace . . . 66
A.5 The discovery curve . . . 66

B Tables, Protocols, and Answers **67**
B.1 Choosing an oracle . . . 67
B.2 Bug class to method . . . 68
B.3 Campaign checklist . . . 68
B.4 Proof-of-concept bundle . . . 69
B.5 First-contact report template . . . 70
B.6 Solutions . . . 71

Glossary **81**

Bibliography **84**

Colophon **91**

Index **95**

CHAPTER 1

The Shape of the Problem

The parser was fine. Everything else about that release went wrong, but the parser was fine.

— Post-incident review, Assurance Team 12, Ninth Street, 2013. The filing engineer was me.

BY THE END OF THIS CHAPTER

State what makes a defect a vulnerability rather than a bug, and name the trust boundary that decision depends on. Apply the three questions of reachability, control and consequence to a candidate defect, and say which of them you have actually answered. Date the field's milestones well enough to know which ideas are old and which are load-bearing. Distinguish, in writing, "there are no bugs here" from "I did not look".

1.1 What a vulnerability is

Software is built in layers that do not trust each other equally, and every layer has an edge. The kernel refuses to let one process read another's pages. A browser puts the renderer in one address space and the code that can reach your filesystem in a different one. A **setuid** binary distinguishes the user who invoked it from the privileges it runs with. A parser separates bytes that have been checked from bytes that have not. Somebody drew each of those lines, and a good many of them were guessing when they did it. What earns the lines your attention is not the judgement of the people who drew them but the sixty years of adversarial testing they have been through since, in public, at scale, with the results written down. Some held. Learning which, and why, is most of what separates this course from a catalogue, because a boundary you merely respect is one you cannot reason about on the day the conditions change, and the conditions always change.

The definition below is the one the rest of the book is built on, and it will be quoted back at you in every chapter:

Definition 1.1 (Vulnerability) *A vulnerability is a* reachable *program state in which behaviour departs from specification, such that the departure is* influenceable *by an actor outside the trust boundary and* consequential *for someone inside it.*

Read it twice. Every word in it is doing work, and three of them are traps.

Reachable: a defect no input can drive the program into is a code-quality problem, not a vulnerability. The trap is that reachability is a fact about your instruments as much as about the program, and Chapter 5 is largely an argument that most of the bugs you fail to find are sitting in code nothing you ran ever executed.

Influenceable: whether a program can be made to crash is rarely the interesting question, and how much of the crash the attacker selected is always the interesting question. A program that dies, and a program that dies in the manner it was instructed to die, are two findings with two prices.

Consequential for someone inside: the trap here is that the phrase contains a variable. Consequence is measured against a trust boundary, and the boundary is not in the code. You write it down, or the vendor writes one down for you and you get back "works as intended". That reply is usually accurate. It is usually also a report of your failure to say, in advance, who was supposed to be outside.

Now the word you arrived with. Nearly everyone turns up thinking of a vulnerability as a *hole*: an absence in an otherwise sound wall, which you locate, widen, and climb through. The picture is comfortable and it costs candidates their first year. A hole is a thing that is missing, so hunting for one means hunting for what is not there, and the only quantity in the whole image is how hard you push.

Very little works that way. The allocator handed out sixteen bytes and the copy loop believes it received thirty-two. Examine either in isolation and neither is defective: the allocator did what it was asked, and the loop does exactly what its bound tells it. What is defective is that two beliefs about one number exist in the same program and nothing reconciles them. The length field says **0x10** and the parser has already consumed **0x20**. A header claims four records, the loop that reads them stops when the bytes run out, and no line of code was ever made responsible for the two counts agreeing.

The word for that is *disagreement*, and I will use it for the rest of the book. It beats *hole* for a practical reason and not an aesthetic one: it tells you what to go and look at. Absences are invisible, and a disagreement has two sides, both present in the source, both written down by somebody, both readable on a Tuesday afternoon.

I kept a folder of them for about fifteen years, beginning as an actual manila folder and ending as a repository, shedding perhaps a third of the contents at each migration.[1] I stopped adding around 2016, and the reason was not that I had collected enough of them. It was that I had stopped being able to tell, on seeing a new one, whether it was new.

[1]The paper ones are the only copies I regret. A disagreement written out by hand in 1999 by the engineer who caused it is a different document from a bug report.

PLATE 1 A lock chamber holds two bodies of water at different levels, and everything downstream depends on a line somebody drew and then built to. The gates are not strong in any absolute sense; they are strong *relative to a stated difference*. Change the boundary — raise the upper pound, open the wrong gate first — and the same timber that was adequate becomes the failure. Consequence in this book is defined the same way, and a boundary you did not write down is a boundary you have left somebody else to draw.

Strength never enters into any of this. It took me something like six years to stop reaching for the hole metaphor, and I still catch myself doing it in front of a class.

A program hardened against every attack its authors could imagine still comes apart in front of a patient reader with a text editor and three free evenings, because the authors defended against the holes they could picture and nobody was ever assigned to check that the program's parts agreed with each other. Chapter 3 teaches the hunt for disagreements instead of holes, and it is the chapter that will get you hired.

1.2 Three questions, restated

You have met the vocabulary of weaknesses in your first-year survey: buffer overflow, use-after-free, injection, race. Catalogues are useful. CWE is a catalogue and a good one, and a catalogue is still not a method. Give a candidate nothing but a catalogue and you get what I see in the first week of every term: they grep for **strcpy**, they bring me seventeen findings, and not one of the seventeen is a vulnerability.

For our purposes there are three questions, asked in order, about every candidate defect. I want to fix them now, informally, before the machinery arrives in Chapter 2.

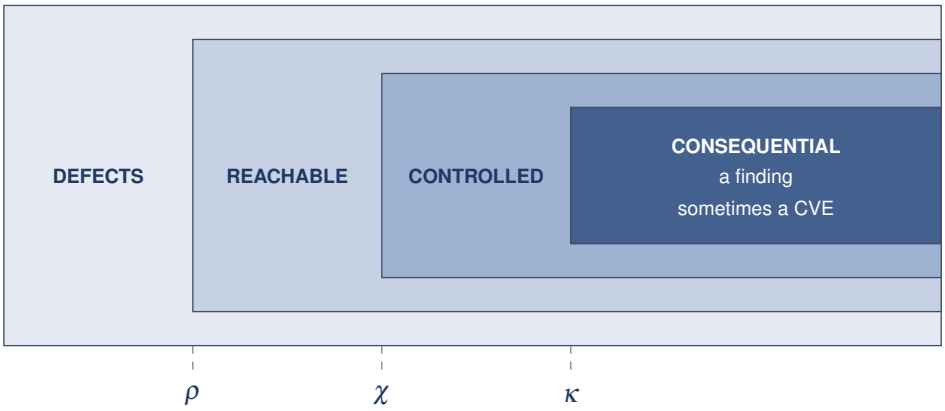


Figure 1-1 The three questions as a filter. Each boundary throws away most of what the one before it let through. The ratios differ wildly from target to target, and measuring your own is the only honest way to estimate how long an engagement will take.

Principle 1.2 (The three questions) *For a candidate defect at program point v: (i)* REACHABILITY *— does there exist an input, consistent with the interfaces an attacker actually reaches, that drives execution to v? Write $\rho(v)$ for the answer, which is in general a path condition and not a yes or a no. (ii)* CONTROL *— of the state at v that matters (the length, the index, the pointer, the type tag, the contents), how much does the attacker select rather than merely perturb? Write $\chi(v)$, measured in bits you can name. (iii)* CONSEQUENCE *— what is the strongest outcome reachable from the corrupted state, given the mitigations actually deployed on the actual target? Write κ, and be honest about it, because κ is the term everyone inflates.*

Everything in this book is a consequence of Principle 1.2. We fuzz because REACHABILITY is hard to answer by reading and easy to answer by running (Chapter 4). We read code because CONTROL is invisible to a crash and legible in a diff (Chapter 3). We write proofs of concept because CONSEQUENCE is a claim, and an unsupported claim is how a real finding gets closed as *won't fix* (Chapter 8). And we stay honest about κ because I have watched what happens to a team that stops. Once every null dereference in your reports is remote code execution, nobody reads your next report. I have been on the receiving end of that judgement and it took two years to undo.

The three questions also explain the shape of Figure 1-1. It is the most useful picture in this chapter, and the reason your first hundred findings will mostly not be findings.

1.3 A short and partial history

The folklore period I have already praised. Let me mark only the milestones a working researcher should be able to date, because each one is a concept you will use this term.

Through 1988. Memory-unsafe code as craft, audited (where it was audited at all) in the vocabulary of correctness and style. The craft record is astonishing; the security record is anecdote. Then in November 1988 the Morris worm propagated in part through a stack overflow in **fingerd**, which had called **gets()** on a network-supplied line, and the industry discovered in a weekend that a memory-safety defect is a remote-code-execution defect. **gets()** was not removed from the C standard library until C11, twenty-three years later,[2] which tells you something durable about the relationship between knowing and doing.

1990. Miller and colleagues at Wisconsin fed pseudo-random input to almost ninety Unix utilities across seven versions of Unix, and crashed between a quarter and a third of them.[3] The paper is short, the tool is a few hundred lines, and it establishes the founding empirical fact of our field: *software fails on inputs its authors never imagined, at rates that embarrass the authors, and you can discover this cheaply.* Every fuzzer in Chapter 4 is an argument about how to spend the cycles that Miller spent uniformly.

1996–2001. The disclosure of technique. Aleph One's stack-smashing tutorial (*Phrack* 49) turned a known bug class into a taught one; the heap papers followed; Solar Designer, Bugtraq, and the full-disclosure wars established both that public technical detail accelerates defence and that it accelerates offence. We are still having that argument, and Chapter 8 will make you take a position on it. The defensive answer arrived in the same window: stack canaries, non-executable pages, and eventually address-space layout randomisation. Each raised the cost of exploitation without touching the rate of defect introduction, a distinction Chapter 9 argues we still do not manage well.

2006. *The Art of Software Security Assessment* systematised manual auditing: attack-surface enumeration, trust-boundary analysis, the arithmetic bug taxonomy, the discipline of reading with a threat model rather than reading for beauty. If this book has a grandparent, that is it.

2011–2013. The instrumentation revolution, and the reason your job is easier than mine was. AddressSanitizer made memory errors *loud* at a published 73% slowdown rather than the order-of-magnitude penalty of the previous generation of tools; and American Fuzzy Lop made coverage feedback cheap, turning fuzzing from a lottery

[2] Which is also about how long it took me to stop reaching for it in throwaway code.

[3] The origin story is better than the paper and I tell it every year: line noise on a dial-up connection during a thunderstorm kept killing the utilities Miller was working in, and he wondered how far that went.

into a search. An oracle that fires on corruption, and a search that climbs coverage: Chapter 4 is an extended argument that the two together constitute the largest single increase in defect-finding productivity in the history of the field.

2016–2020. Industrialisation. Continuous fuzzing services took the campaign out of the researcher's basement; by mid-2025 Google's OSS-FUZZ reported having helped identify and fix over 13,000 vulnerabilities and 50,000 bugs across a thousand projects. In the same period, static analysis grew a query language, and with it variant analysis: the practice of turning one bug into a pattern and the pattern into a sweep across every codebase you can index. It is now a first-class method, and it is §3.6.

Since. What will shape your career more than anything in Chapters 2 through 7 is still going on around you. Memory safety at the language level, first: the proportion of Android's vulnerabilities attributable to memory safety fell from 76% in 2019 to 24% in 2024, in absolute terms from 223 to fewer than fifty, as new code moved to Rust and Kotlin. Not by finding the bugs faster. By not writing them.[4] Automated discovery by language models, second: in 2024–2026 agentic systems began reporting real, non-trivial memory-safety bugs in SQLITE, OPENSSL, and the Linux kernel, including at least one that a vendor characterised as known to attackers before it was known to defenders. Chapter 9 discusses what this does to the economics of our field, and declines to tell you it will be fine.

1.4 Why the method is not optional

Some of you are wondering whether all this apparatus is really necessary, given that the tutorial you read at sixteen got you a shell. It is a fair question.

First: the tutorial had a failure rate you never saw. It worked on the one target it was written for. You do not have the luxury of an unrecorded failure rate: your failures will consume weeks that somebody is paying for, and the person paying is entitled to know whether the two weeks bought coverage or bought nothing. Method is what lets you tell "there are no bugs here" apart from "I did not look". In a badly written report those two findings are the same sentence. They are not the same fact, and the person paying you is entitled to know which one they bought.

Second: you are not auditing against an absence. You are competing against other researchers, against vendors' own programmes, and against organisations whose budget

[4] Google's own framing is worth learning properly: the mechanism is not that Rust code is audited harder but that the vulnerability *density* of new code collapses, so the vulnerable fraction of a growing codebase decays even with no rewrite of the old parts. If you plan a thirty-year career in C auditing, plan a second career too.

for this exceeds your employer's revenue. The bugs findable by grep are found. What remains is what remains *because* it requires a model: the integer that overflows only on 32-bit builds, the state machine that permits a second **free** only after an error path, the two parsers that agree on every input a human would generate. Against a well-fuzzed target, an unsystematic search is a coin flip you pay salary for.[5]

Third, and this is the real answer: because the method is where the honesty lives. Our output is other people's decisions about what to patch first, which makes calibration the product rather than a virtue attaching to it. A crash tells you almost nothing. Enthusiasm will fill the gap with a story, the story is always *attacker-controlled write, probably code execution*, and the story is usually wrong. The apparatus in this book exists to convert enthusiasm into claims with evidence attached, so that when you finally write *this is exploitable*, the sentence means something, because you have spent four years being the sort of person who also writes *this is not*.

RANGE SAFETY

Nothing in this chapter authorises testing. Candidates hold no testing authorisation against any system they do not personally own until the VR-201 written examination is passed and a scope document is countersigned. The Institute's disciplinary record contains fourteen years of candidates who felt that a university web application was surely fair game, and the relevant statutes in most jurisdictions turn on *authorisation*, not on intent, not on harm, and *not on whether you told them afterwards*. Own the target, or hold written permission naming the target, the dates, and the techniques. There is no third category, and the second category is a document, not a conversation.

NOTES AND SOURCES

The Morris worm is described in Spafford's contemporary analysis [48]; the worm propagated by several routes, of which the **gets()** overflow in **fingerd** was one. **gets()** was removed from the language in C11 [25]. Miller, Fredriksen and So [30] tested almost ninety utilities across seven versions of Unix and crashed between a quarter and a third of them; their own paper notes that one of the bugs they found shared its root cause with the worm's. That is a neater link between these two milestones than the chronology suggests. The stack-smashing tutorial is Aleph One's [1]; the defensive answer of the same period is represented here by StackGuard [15]. The manual-auditing tradition this book descends from is [16].

AddressSanitizer's published cost is a 73% slowdown and 3.37× memory on SPEC CPU2006 [45]; the previous generation's order-of-magnitude penalty is Valgrind's [32]. OSS-Fuzz's cumulative figures are the project's own [35]. The Android proportions are Google's [21], which also makes the density

[5] A colleague objects that a coin flip is fairer. He is right, and I have never found a better line to replace it with.

argument in the footnote: the mechanism is that new code stops introducing the defects, not that old code is audited harder. On agentic discovery, the SQLite case is CVE-2025-6965 [22] and the kernel case is CVE-2025-37899 [24]; note that the "known only to threat actors" characterisation is the finder's, not the vendor's.

Exercises

Starred exercises require a licensed range, a written scope, and a supervising Second Bench. Unstarred exercises require a text editor. Do not confuse the two.

Exercise 1-1. Take Definition 1.1 and identify the trust boundary, the reachable interface, and the consequence class for: (a) a command-line image converter run by a user on their own files; (b) the same converter invoked by a mail gateway on attachments; (c) the same converter's library, linked into a browser. The code is identical in all three. Explain, in one page, why the same defect is a non-issue, a serious finding, and an emergency, and why the vendor's answer to your report will differ in each case.

Exercise 1-2. A colleague reports "I fuzzed it for a week and found nothing, so it's secure." Rewrite that sentence three times: once as an honest statement of what was tested, once as an honest statement of what was *not*, and once as the single question you would ask to determine which of the two the week actually produced. Keep your answer; you will grade it again after §5.1.

Exercise 1-3. For each of the following, state whether it is a defect, a reachable defect, a controlled reachable defect, or a finding, and name the fact you would need to establish to move it one level right: (a) an assertion failure in a debug build; (b) a null dereference in a request handler; (c) a one-byte heap overflow with a fixed value; (d) a one-byte heap overflow with an attacker-chosen value; (e) an uninitialised read whose result is returned to the caller.

Exercise 1-4. List three interfaces of a program you use daily that accept input from someone who is not you, and three that do not. At least one entry in each list must be something you had not previously thought of as an interface. The point of the exercise is the habit; the habit is attack-surface enumeration; and the habit is the profession.

Exercise 1-5. Find a fixed, public memory-safety CVE in a project you have never read, locate the patch, and write one paragraph on what the *two disagreeing beliefs* were, in the sense of §1.1: who thought what about which value. Do not read the finder's write-up until you have written the paragraph. Then read it, and note in a second paragraph what you missed. Repeat weekly for the rest of the course; this exercise, done fifty times, teaches more than this book does.

CHAPTER 2

The Machine Underneath

The program you wrote is not the program that ran. Reason about the one that ran.

— T. Vance, *Fault Injection* (1996), §1.

BY THE END OF THIS CHAPTER

Hold a working model of memory that predicts real failures: objects with size, alignment, type and lifetime, and the two safety properties that memory-safety bugs violate. Explain why undefined behaviour is a licence to the optimiser rather than a promise of a trap, and why that means the source is a hypothesis about the binary. Choose the cheapest oracle that detects a given bug class, and, more important, say what each oracle cannot see.

2.1 What we model, and what we do not

Everything in this book that concerns C and C++ rests on a mental model of memory, and the model has to be accurate enough to predict real failures while staying crude enough to hold in your head at three in the morning. This chapter builds that model. It is *engineering-true*: it predicts essentially every memory-safety phenomenon you will meet in this course, and it is known to be wrong in ways Chapter 9 catalogues honestly. The genuinely correct model (the C abstract machine, with pointer provenance, effective types, and the full apparatus of the standard) exists, matters enormously to compiler authors, and is somebody else's course.[1]

Hold the model loosely. Candidates turn up wanting one true picture of memory and are put out to find there are four of them: the one in the source, the one in the compiler, the one in the allocator, and the one in the silicon.[2] Worse, a great many vulnerabilities are a disagreement *between* two of the four, and that is not a defect of the field but the field.

2.2 Objects, lifetimes, and the two safeties

A C object is a region of storage with a size, an alignment, a type, and a *lifetime*. Almost every memory-safety bug you will ever find violates the size or the lifetime, and the two are worth naming separately because they are found by different means and detected by different tools.

[1] VR-405, *Language Semantics for Assurance*, taught by a colleague with more patience for the aliasing rules than I have ever possessed. Take it in your final year, after you have been personally injured by the optimiser at least once.

[2] Five if you count the one in the documentation, which is usually a fossil of the one in the compiler as it stood two rewrites ago.

Definition 2.1 (Spatial and temporal safety) *An access is* spatially *safe if it falls entirely within the bounds of the object the pointer was derived from. It is* temporally *safe if it occurs within that object's lifetime. A memory-safety violation is a failure of one or both.*

Spatial failures are the overflow family: buffer overflow and underflow, out-of-bounds read and write, off-by-one, index confusion, and intra-object overflow: writing past a struct member into its sibling. That last is a real bug, and, as §2.5 notes, one AddressSanitizer does not catch. Temporal failures are the lifetime family: use-after-free, double free, use-after-return, use-after-scope, and the dangling pointer nobody freed. A **realloc** moved the allocation; a **std::vector** reallocated while you held an iterator into it.

The third category is *type confusion*: an object accessed through a pointer to an incompatible type, so that the bytes are in bounds and in lifetime but are interpreted as the wrong thing. It is less discussed than the other two and disproportionately represented among bugs that survive years of fuzzing. A field that is an integer in one view and a pointer in the other is an address-disclosure primitive read one way and a write primitive the other, and that is why type confusion in scripting engines and document interpreters is worth what it is worth.

2.3 Undefined behaviour, and the term that changes everything

If C merely had unsafe operations the model would be simple: unsafe operation, unsafe result, done. But C has *undefined behaviour*, and that difference is the entire character of the language from a security standpoint.

Principle 2.2 (The optimiser's assumption) *Where the standard leaves behaviour undefined, an implementation may assume it does not occur. The compiler is therefore licensed to delete, reorder, or reason around any code path reachable only if undefined behaviour has already happened.*

Think about what that licenses. Consider the check every candidate writes in their first week:

```
/* Intent: reject the case where the addition wraps. */
int add_checked(int a, int b, int *out) {
    int sum = a + b;
    if (sum < a)            /* survives -O2; see below */
        return -1;          /* overflow, we hope */
    *out = sum;
    return 0;
}
```

Signed integer overflow is undefined. Therefore **a + b** may be assumed not to overflow, and therefore **sum >= a** *whenever* ***b >= 0***. Read that condition carefully. The compiler does not know the sign of **b**, so it cannot fold the comparison, and it does not: build this on **gcc** and **clang** at every optimisation level and the branch is present in all of them. Candidates are routinely taught otherwise, and the disassembly is the only thing that settles it.

Now supply the missing premise. Narrow **b** to an unsigned type, so the compiler knows what the standard already let it assume:

```c
/* Same check. Now the compiler knows b >= 0. */
int add_checked(int a, unsigned short b, int *out) {
    int sum = a + b;
    if (sum < a)            /* eliminated from -O1 upwards */
        return -1;
    *out = sum;
    return 0;
}
```

At **-O1** and above, on both compilers, the branch ceases to exist. The programmer wrote a check; the binary contains none; an audit that reads the source and stops there records a control that is not deployed. The same happens to the classic pointer-overflow guard **if (p + n < p)**, which is eliminated from **-O1** upwards because pointer arithmetic past the end of an object is undefined too.

Undefined behaviour licenses the optimiser but does not compel it, so *whether* a check survives is a property of what the compiler could prove on the day, and the version that looks safer is the one that got deleted. Check the binary. §8.3 makes a discipline of it: you have not understood a bug until you have looked at what the compiler emitted.

Corollaries, and these you should memorise:

1. **Check before, not after.** Test operands against the limit (**b > INT_MAX - a**), never the result against an operand. For sizes, prefer checked builtins such as **__builtin_add_overflow** / **__builtin_mul_overflow**, or an explicitly unsigned computation with an explicit bound.
2. **Unsigned wraps; signed does not.** Unsigned arithmetic is defined to wrap modulo 2^n: not undefined, merely wrong. Unsigned overflow bugs therefore survive optimisation and are found by reasoning; signed overflow bugs are found by UndefinedBehaviorSanitizer and by reading disassembly.
3. **The source is a hypothesis.** Every claim of the form "the code validates *X*" is a claim about the binary. Know which of your claims you have actually checked.
4. **The optimisation level is part of the finding.** A check that exists at **-O0** and not at **-O2** makes those two builds different programs, and a report that does not name the build is not reproducible by anyone. Chapter 8 makes the build identification a required field of the artefact for exactly this reason.

The arithmetic in real bugs is almost never interesting arithmetic. It is **n * sizeof(elem)** where **n** is attacker-supplied and the product wraps; a **size_t** length assigned to an **int** and compared against a bound that is now negative; **len - 1** where **len** may be zero; a signed **char** promoted to **int** and used as an index. Chapter 3 catalogues these properly. What Chapter 2 asks you to internalise is only that the language answers arithmetic error by licensing the optimiser.

2.4 Where the objects live

The stack

A call frame holds saved registers, the return address, spilled locals, and any arrays the function declared. Its two security-relevant properties are that it is *adjacent* — a linear overflow of a local array runs into other locals, then saved registers, then the saved return address — and that its layout is a compiler decision. Modern compilers reorder locals to place arrays nearest the canary: a mitigation, not a guarantee.

Figure 2-1's caption states the lesson; this paragraph is here because the lesson takes about a term to absorb. Everyone's first instinct is that a stack overflow matters because it reaches the return address. On a

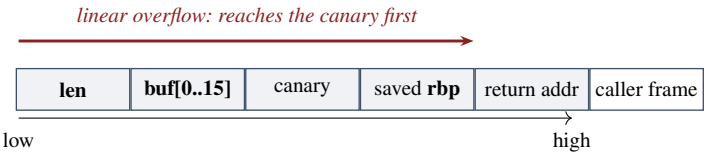


Figure 2-1 A conventional x86-64 frame, with locals reordered by the compiler as §2.4 describes: the array sits nearest the canary, scalars below it. The mitigation therefore stands between the array and the return address, and a linear overflow reaches the canary before anything worth having. The corollary is the useful one — an *under*flow, or an attacker-chosen negative index, runs the other way into **len** and the other scalars, where nothing guards them. The most valuable thing a small overflow reaches is rarely the return address; it is a length or a pointer that widens the *next* write.

modern target the return address is the *least* useful thing it reaches, because it is guarded; the useful thing is the adjacent length, the adjacent flag, the adjacent pointer, because corrupting those converts a small deterministic overflow into a large controlled one. Look for the second write, not the first.

The heap

A heap allocator stores its own metadata — sizes, free-list links, bin indices — in the same address space as the data it hands out, usually immediately adjacent to it. That single design decision originates an entire bug class, and the mitigations of the last two decades (mangling and safe-linking of free-list pointers, size-class isolation, out-of-line metadata, delayed reuse, and hardened allocators such as Scudo, hardened_malloc, and PartitionAlloc) are attempts to buy back what it costs.

For *finding* bugs, these allocator facts suffice:

- **Adjacency is influenceable.** Which object sits after yours is a function of allocation order and size class, and allocation order is frequently a function of input. "The overflow only reaches four bytes of padding" is a claim about one run.
- **Reuse is prompt.** Most allocators hand a freed chunk to the next request in the same size class almost immediately. A use-after-free is therefore in practice a type-confusion primitive, and its severity depends on what else the program allocates at that size — which is, again, input-influenced.
- **The allocator is an oracle, and a late one.** Aborts such as "double free or corruption" or "invalid next size" are the allocator noticing damage that already happened, at a place that is not where the bug is. Chapter 8 is largely about that gap.

2.5 Instruments

You will not find these bugs by staring. You will find them because you arranged in advance for the machine to tell you. The largest single determinant of how many bugs a campaign finds is the *oracle* you compiled in.

Definition 2.3 (Sanitiser) *A sanitiser* Σ *is an instrumentation-plus-runtime pair that converts a class of silent undefined behaviour into a loud, immediate, attributable abort.*

AddressSanitizer (ASan) is the workhorse, and its mechanism is worth knowing exactly, because

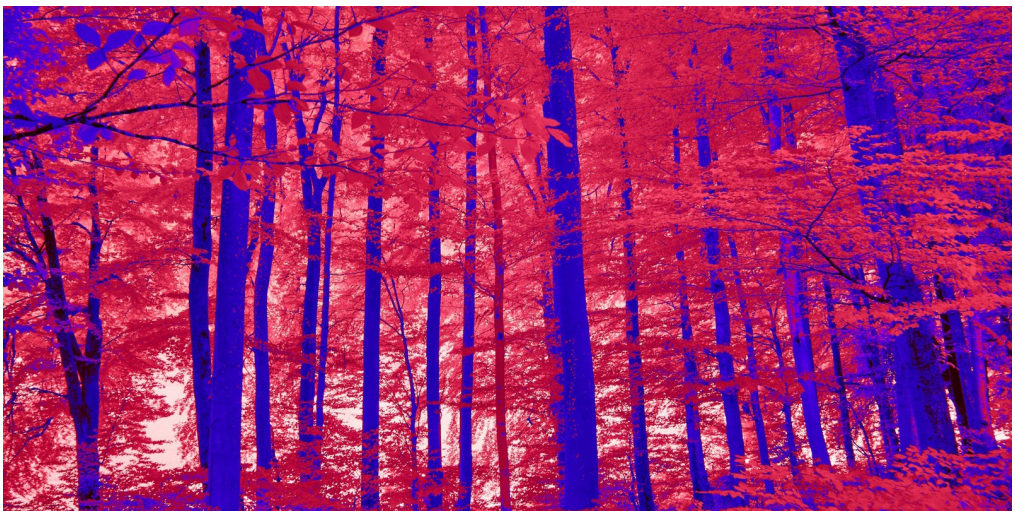


PLATE 2 Reproduced in colour, alone among these plates, because the colour is the point. Nothing here is hidden: the wood looks like a wood. The magenta is the *instrument's* output rather than a property of the scene, and it separates living foliage from dead wood at a glance because the sensor answers a question the eye does not ask. A sanitiser does the same work on a running program. The corruption was already there, silent and, as far as the processor was concerned, perfectly in order; the instrument is what makes it loud enough to attribute.

knowing it tells you what it cannot see. Every eight bytes of application memory map to one byte of *shadow* memory at $(\text{Addr} \gg 3) + \text{Offset}$; that shadow byte records how many of the eight are addressable. The compiler inserts a shadow check before memory accesses. The runtime replaces the allocator so every heap object is surrounded by poisoned *redzones* (16 bytes minimum, scaled up to 2048 for large allocations), poisons freed regions, and holds freed chunks in a *quarantine* to delay reuse.[3] Stack objects get redzones too; globals get them at link time. The published cost is a 73% slowdown (1.73×, the figure §B.1 tabulates) and 3.37× memory on SPEC CPU2006; budget 2–3× time in practice, and expect worse memory on allocation-heavy targets. The same paper reports 24.55× memory on one of its own benchmarks, so treat any single quoted figure as a property of the workload rather than of the tool.

Now the part candidates skip and should not:

- ASan finds heap, stack, and global overflows; use-after-free; use-after-return and use-after-scope with the relevant options; double free; invalid free.
- ASan does **not** find uninitialised reads. That is MemorySanitizer, which requires the C++ standard library to be instrumented too, and is correspondingly more work to stand up. The runtime ships interceptors for the common libc entry points, but **libstdc++** has to be replaced with an instrumented **libc++**.
- ASan does **not** reliably find *non-linear* overflows. A write at **base + attacker_index** with a large index steps clean over the redzone and lands inside another live object, where the shadow says "addressable" and nothing fires. This is not a corner case; it is the shape of a great many real bugs, and it is why §4.5

[3] Quarantine is the only part of ASan whose name I have never had to explain to anybody.

asks you to run sanitiser builds *and* read code.

- ASan does **not** find intra-object overflows: writing past **s->name[16]** into **s->flags** is, to the shadow map, an ordinary access within one allocation.
- Quarantine is finite. Exhaust it and a use-after-free becomes a use-after-reuse. That one is silent.

The other instruments, briefly, with the vocabulary fixed. **UBSan** (**-fsanitize=undefined**) traps the undefined arithmetic and pointer behaviours of §2.3 and is cheap enough to leave on nearly always; note that **unsigned-integer-overflow** is *defined* behaviour and so is a separate, noisier opt-in check. **MemorySanitizer** finds uses of uninitialised memory. **ThreadSanitizer** finds data races: the bug class most under-hunted relative to its severity, because races are invisible to every deterministic technique in Chapters 3 and 4. **Valgrind/Memcheck** needs no recompilation and is therefore the right tool for a binary you cannot rebuild,[4] at a slowdown measured for this edition at between six and fourteen times, depending on how much of the work is memory traffic. Origin tracking costs roughly half as much again. **KASAN**, **KMSAN** and **KCSAN** are the kernel equivalents, and **kcov** is the kernel's coverage interface; together they are what makes §7.6 possible at all.

WORKED EXAMPLE 2-1 The report you will read most often

An ASan report is a structured document, and candidates skim it. Read it in this order. *First line*: the error class (**heap-buffer-overflow**, **heap-use-after-free**, **stack-buffer-overflow**) and **READ** or **WRITE** of size n. A **WRITE** of size 1 and a **WRITE** of size 8 are different findings; a **READ** is usually a disclosure question and a **WRITE** a corruption question. *Second*: the access offset relative to the object, "8 bytes to the right of a 24-byte region". That number is your first estimate of χ: a fixed small positive offset is an off-by-one; a large or varying offset means an index you may control. *Third*: the allocation stack, which tells you where the object was created and therefore where its *size* was decided. The size decision is very often the bug, and the access very often is not. *Fourth*, for use-after-free: the **freed by** stack, which together with the allocation stack gives you the object's whole life in three traces. Do not open a debugger until you can state, from the report alone, which of those four numbers you would need to change to make the bug worse. ▲

RANGE SAFETY

Sanitiser builds are for your machine and your lab. Two cautions that have each cost a cohort a bad afternoon. *One*: ASan is not a hardening measure and its default configuration is not safe to expose — it maps enormous shadow regions and its reports disclose addresses and paths, so a target built with it should never be reachable from a network you do not control. *Two*: a sanitiser abort is a denial of service you have just built. If you are authorised to test a service, you are testing a *copy*, with its own data, on its own host, and

[4] Which is more often than the phrase suggests. Vendor toolchains go out of support, and the build that shipped is not always a build anyone can reproduce.

> the scope document says so in writing. Instrumenting production because "it will only abort if there's already a bug" is a sentence that has ended careers, and its circularity is obvious to everyone except the person saying it at two in the morning.

NOTES AND SOURCES

The spatial and temporal vocabulary, and the taxonomy behind it, are set out in [51]. Undefined behaviour is defined at ISO/IEC 9899 §3.4.3, unsigned wraparound at §6.2.5, and pointer arithmetic at §6.5.6 [25]; for what optimisers actually do with the licence, see [53].

The claim in §2.3 that the first listing's check survives is not an argument but a measurement: both listings were compiled on GCC 15.2 and Clang 21.1 at every optimisation level, and the results are as stated. AddressSanitizer's shadow mapping, redzones and quarantine are described in [45]; the current defaults — a 16-byte minimum redzone, a 2048-byte maximum, a 256 MB quarantine, and stack-use-after-return detection off unless enabled — are from the project wiki [43], checked July 2026. Intra-object overflow is outside ASan's default reach; an experimental Clang flag adds padding between fields of C++ *classes* and catches some cases [43], which does not help the C struct in the example here. MemorySanitizer's 3× figure and its instrumentation requirement, ThreadSanitizer's 5–15×, and UndefinedBehaviorSanitizer's check set are from the Clang documentation [14].

Exercises

Starred exercises require a licensed range, a written scope, and a supervising Second Bench. Unstarred exercises require a text editor. Do not confuse the two.

Exercise 2-1. Compile both versions of **add_checked** at **-O0** and **-O2** on two compilers, disassemble all eight builds, and state for each whether the check exists. Predict the answers before you look. Then rewrite the function so that it is correct *and* its check survives every build (the checked builtins of Corollary 1 are the short route), and verify that it does. Finally, state in one sentence why this exercise, and not any amount of reading, is the reason to trust your rewrite.

Exercise 2-2. A 24-byte heap object is followed in memory by a 24-byte object of another type. Explain why "the overflow only corrupts padding" is not a property of the bug, and name two things an attacker can vary to change what follows.

Exercise 2-3. For each bug class name the cheapest oracle that detects it and one realistic situation in which that oracle stays silent: (a) a 4-byte heap overflow; (b) a heap write at index 2^{20}; (c) use-after-free with an exhausted quarantine; (d) a read of an uninitialised struct field; (e) signed overflow in an index computation; (f) a data race on a reference count.

Exercise 2-4. Take any struct in a real codebase containing a fixed-size array followed by a non-array member. Explain why an overflow of the array is invisible to ASan, propose a change to the *struct* — not to the code that writes it — that would make the bug detectable, and then explain why the maintainers will decline your change.

Exercise 2-5. * (Supervised; own machine only.) Build a small target three ways — no sanitiser, ASan, ASan+UBSan — and measure wall-clock throughput over a fixed corpus of a thousand inputs. Record

executions per second for each. You are not fuzzing yet. You are learning the exchange rate between *oracle strength* and *executions*, because for the rest of your career you will spend one to buy the other, and §4.5 will ask you to price the trade without looking it up.

CHAPTER 3

Statics: Reading Code for Bugs

Given the boundary, the interior follows. There is no second interior.

— D. Prather, lecture notes, Ninth Street, 2003.

BY THE END OF THIS CHAPTER

Read a codebase backwards from its dangerous operations and forwards from its inputs, and recognise the place in the middle where a guard is missing, checks the wrong variable, checks at the wrong time, or checks in the wrong type. Identify the six arithmetic shapes on sight. Turn a single defect into a pattern and the pattern into a sweep. And produce, as the output of an audit, a document rather than an impression.

3.1 The audit problem

Fix a codebase P and a trust boundary $\mathscr{T}$. The audit problem is: given the interfaces an attacker can reach across $\mathscr{T}$, and the operations inside P that can be made to misbehave, does there exist a path from the former to the latter that fails to establish, along the way, every precondition the operation requires? The chapter's answer, in one sentence: a *complete* chain of checks makes a sink safe, and every bug you will ever be paid to find is a failure of the word *complete*.

That framing already tells you how to read. You do not read a codebase front to back. You read *backwards from dangerous operations* and *forwards from attacker-controlled inputs*, and you look for the place in the middle where the two meet without a check between them.

3.2 Sources, sinks, and the completeness argument

Definition 3.1 (Source, sink, guard) *A* source *is a program expression whose value is, wholly or partly, chosen by an actor outside $\mathscr{T}$. A* sink *is an operation with a non-trivial precondition on its operands — a memory write, an index, an allocation size, a length, a free, a type cast, a command. A* guard *is a check that establishes a sink's precondition on the specific value that reaches it.*

Theorem 3.2 (Completeness, informal) *Let k be a sink with precondition φ. If every feasible path from every source to k passes through a guard that establishes φ* for the value that arrives at *k, then k cannot be driven out of specification from outside $\mathscr{T}$. If a single feasible path lacks such a guard, then the states reachable at k form a family whose member is selected by the attacker.*

Sketch. The forward direction is a path induction over the control-flow graph: if φ holds on entry to k on every path, it holds at k. The converse is the interesting half, and its content is entirely in the phrase *for the value that arrives*. The implication fails in practice in these ways, each a bug class: the guard checks a *different* variable; the guard checks the *same* variable at a different *time*, and the value changes in between;

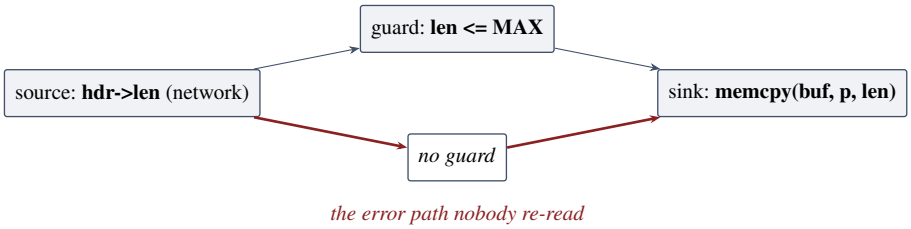


Figure 3-1 Completeness is a property of the path set. The upper path is correct and will be the one in the documentation, in the code review, and in the maintainer's memory. The lower path is the finding, and in real codebases it is almost always an error path, a retry path, a legacy entry point, or a second caller added three years later.

the guard checks the same variable at the same time but in a different *type*, and the conversion is not order-preserving. □

Theorem 3.2 is not novel and carries nobody's name. It went under one in earlier printings; see the notes. What is worth keeping is the phrase *for the value that arrives*, and the three failure modes in the proof sketch.

Corollary 3.3 (The no-back-door corollary) *The strength of the guards on the checked paths is irrelevant to the existence of the unchecked one. Only completeness closes the family, and completeness is a property of the* set *of paths. Grep finds guards; only reasoning finds their absence.*

Read the corollary the way a locksmith reads it, and read Figure 3-1 alongside it: the guarded path is the one everybody remembers, and the finding is on the other one. The Institute's own finding archive, counted over the last decade, runs to roughly three parts missing-or-wrong guard, one part guard-present-but-defeated (the three failure modes in the proof), and a remainder of logic errors that no amount of memory-safety thinking would have caught. That shape tells you where the reading time goes. It goes on the provenance of the operands, because the dangerous operation itself is trivially easy to find and its operands are not.

EDITORIAL NOTE

The Institute and its archive are inventions of this book's framing, and the proportions above are the author's impression rather than a measurement — they are given as a shape, not as data, and no percentage should be quoted from them. Published root-cause breakdowns do exist, they are worth reading, and they disagree with each other in ways that are themselves instructive. The claim that survives without any of them is the qualitative one: guards are easier to find than the provenance of the values reaching them, so that is where reading time goes.

3.3 The arithmetic that lies

Most memory-safety bugs are arithmetic bugs wearing a memcpy. There are six shapes worth learning cold. Each gets a real case below, because a shape is easy to nod at in a textbook and hard to see in four thousand lines you did not write, and the only cure I have found for that gap is having read the real thing once.

(i) Multiplication overflow in an allocation size

```
n = read_u32(input);                    /* attacker chooses n */
buf = malloc(n * sizeof(struct rec)); /* wraps if n > SIZE_MAX/sizeof */
for (i = 0; i < n; i++) parse_rec(&buf[i], input);
```

The allocation succeeds, small; the loop writes *n* records. The guard, if any, is on **n** against a semantic limit rather than against the arithmetic one. “No real file has forty million records” is a statement about files, and what arrived is not a file but whatever the sender chose to call one.

The case. Stagefright, CVE-2015-1538. Android’s media framework parsed the **stsc** atom of an MP4, took a 32-bit entry count out of the atom, and multiplied it by the size of the per-entry structure to size the allocation. The multiplication was unchecked and the product wrapped. Entries were then written according to the count that had not wrapped, off the end of a heap object, inside a process holding the microphone and camera groups. What made it notorious was the delivery: on the Android of the day an MMS was parsed before anyone opened it, so the user-interaction count was zero.

What closes it. Check the operands, never the product: **n > SIZE_MAX / sizeof(struct rec)** *before* the multiply, or **__builtin_mul_overflow**, or **calloc**, whose overflow behaviour is mandated by CERT MEM07-C and universal in mainstream libcs, though not compelled by the C standard itself and historically not guaranteed: **glibc** 2.2.5 and two Visual C++ releases shipped **calloc** implementations that wrapped silently.

What finds it. A fuzzer, readily, once the harness reaches the parser; Stagefright was found with AFL. UBSan does not, because **size_t** multiplication is unsigned and therefore defined rather than undefined, and the **unsigned-integer-overflow** check that would catch it is a separate opt-in for the reason §2.5 gives. Reading finds it fastest of all. Grep for *** sizeof** and read every hit.[1]

(ii) Truncation across widths

A **size_t** length assigned to an **int**, an **int** to a **short**, a 64-bit offset to a 32-bit index. The guard runs on the wide value; the sink uses the narrow one. This class is why the phrase “only on 32-bit builds” appears in so many advisories, and why you should always ask which build the reporter tested.

The case. Sequoia, CVE-2021-33909, in the Linux kernel’s **seq_file** layer. **seq_file** tracks its buffer size in a **size_t** and then hands that size to code taking an **int**. An unprivileged local user who creates, mounts and deletes a directory structure whose total path length runs past a gigabyte drives the size beyond what an **int** holds, and the ten-byte string **"//deleted"** is written about two gigabytes below the start of a **vmalloc**’d kernel buffer. Qualys turned that into root on default installations of five distributions. The commit that

[1] An afternoon, on most codebases. Less if the project has a house macro for it, which is itself worth knowing about before you start.

introduced it landed in 2014 and the disclosure was 2021. Seven years, in the most-read C codebase on earth.

What closes it. One integer type for a length, chosen at the boundary and carried the whole way down. Where a narrowing conversion is genuinely required it is a *checked* conversion with a guard of its own, and it earns a comment naming the range it enforces.

What finds it. Very little, cheaply. Reaching Sequoia takes roughly five gigabytes of memory and a million inodes, so no ordinary campaign was going to arrive there by mutation. What finds this class is reading the types, plus **-Wconversion** on a codebase kept clean enough that its output is readable rather than a wall.

(iii) Signedness confusion

A length declared **int**, checked with **if (len > MAX) return -1;**, then passed to something taking **size_t**. A negative **len** passes the check and becomes enormous at the sink. The canonical smell is a signed type in a variable whose name is **len**, **size**, **count**, or **n**.

The case. wget before 1.19.2, CVE-2017-13089. The chunked-transfer parser read each chunk length with **strtol** and never checked the result for being negative. A malicious server replying with a non-200 status reaches **http.c:skip_short_body**, which tries to consume the body in 512-byte pieces via **MIN** and passes the negative length down to **fd_read**. **fd_read** takes an **int**, the high bits are discarded, and the length is then entirely the server's to choose. A stack-based overflow, in the program whose output half the internet pipes into a shell.

Notice that two shapes are stacked: the missing sign check is (iii) and the narrowing into **int** is (ii). Real bugs are rarely one shape, and a root-cause note that names only one of them is a note that will not prevent the next instance.

What closes it. strtol returns a signed value, so the guard is **< 0** as well as **> MAX**, and both belong immediately after the conversion rather than down at the sink. A length that cannot be negative should not be stored in a type that can.

What finds it. Reading, mostly. Triggering it requires a hostile *server*, so the harness has to drive the far side of a connection, and §5.3 is largely about why that harness does not exist for most network clients.

(iv) Subtraction below zero

remaining = end - cur; where **cur** may have advanced past **end**; **len - 1** where **len** may be zero; **avail - needed** in unsigned types. Every unsigned subtraction is a potential 2^{64}, and the check that prevents it is a check on the *ordering* of the operands. People omit that check, because ordering feels like an assumption about the world rather than a precondition on an operation.

The case. Lhasa, CVE-2016-2347, an LHA archive extractor. The level-3 header parser read a 32-bit **header_len** from the file and computed the bytes still to read as **header_len - RAW_DATA_LEN(header)**, subtracting the fixed 32-byte part it had already consumed. An archive declaring a header length below 32 underflows the subtraction and the read length becomes something near **UINT_MAX**. The extractor then reads that many bytes out of the file and into a heap buffer.

What closes it. if (header_len < LEVEL_3_HEADER_LEN) return error; above the subtraction rather than below it. Generally: every unsigned subtraction in a parser wants the comparison that proves it non-negative sitting directly over it, where the next reader cannot miss the pairing.

What finds it. A fuzzer, almost at once, because the trigger is a small value in a header field and mutation produces small values in header fields all day. Which tells you something when you meet this

shape in a codebase that claims to be fuzzed: either the harness never reached the parser, or the campaign was never actually run.

(v) Off-by-one at the terminator

The whole **strcpy/strncpy/snprintf** family, the boundary between "length" and "buffer size", and the single most productive question in manual auditing: *does this count include the null byte?* Ask it of every length in every parser you read. You will be right about one time in forty, which is a very good rate.[2]

The case. Netatalk before 3.2.1, CVE-2024-38440. The DHX authentication path took a length field out of an **FPLoginExt** request, failed to validate it against what had been allocated, and wrote a single **0** byte one past the end of a heap buffer. One byte, one value, no choice of content. On its face the least interesting bug in this section, and it is here because that byte lands in allocator metadata. How much of the value an attacker chooses is not the only thing that decides whether a bug matters. Where it lands does too.

A second case, worth knowing because the whole error is visible in one line. GHOST, CVE-2015-0235, in **glibc**'s **__nss_hostname_digits_dots**. The size was computed as **sizeof (*host_addr) + sizeof (*h_addr_ptrs) + strlen (name) + 1**: room for two structures, the name, and the terminator. The code then wrote a third pointer, **h_alias_ptr**, whose size nobody had added to the sum. The overflow is four bytes on 32-bit and eight on 64-bit, made of digits, dots and null bytes, and Qualys still got remote code execution out of it against Exim.

So the question is not only whether the count includes the null. It is whether the sum includes every single thing you are about to write.

What closes it. Write the size as a sum with one term per object written, in the order they are written, and put the terminator in the sum explicitly. Sizes expressed as a formula are where this class lives. Sizes expressed as a list are where it does not.

What finds it. ASan, the moment the path executes. That is most of the argument for building with it. Note that both cases above are on the heap. The identical mistake between two members of one struct is an intra-object overflow, and ASan is blind to it (§2.5).

(vi) The size decided once and used twice

A size computed for allocation and recomputed, or assumed, at use, with the two computations diverging on some input. This is the deepest of the six and the hardest to fuzz for. It has the two case studies of §3.4 to itself, and the worked example below is a small instance of the same thing.

WORKED EXAMPLE 3-1 Auditing a length-prefixed record parser

Here is a fragment of the kind you will read a thousand times. Find the bugs before reading on; there are three.

[2] I have never kept count, so that is a feeling with a number attached to it, which is the thing the rest of this book tells you not to do. I have left it in because striking it would be worse: you would assume I had measured it.

```
int parse(const uint8_t *in, int in_len) {
    int off = 0;
    while (off < in_len) {
        uint16_t type = rd16(in + off);
        int      len  = rd16(in + off + 2);
        char     name[64];
        if (len > sizeof(name))            /* (1) */
            return -1;
        memcpy(name, in + off + 4, len);   /* (2) */
        name[len] = '\0';                  /* (3) */
        handle(type, name, len);
        off += 4 + len;                    /* (4) */
    }
    return 0;
}
```

First, line (1) is an off-by-one: **len == 64** passes, and line (3) writes the terminator at **name[64]**, one past the array. That is a one-byte stack overflow with a fixed value — low χ, but on the wrong compiler layout it is adjacent to something that matters, so it is a finding. *Second*, and worse, nothing establishes that **off + 4 + len** is within **in_len** before line (2). The guard bounds **len** against the *destination* and never against the *source*. That is an out-of-bounds read of up to 64 bytes past the input buffer, disclosed to **handle**. Bounding a length against the destination while forgetting the source is the most common single error in this genre. *Third*, the loop reads **rd16(in + off)** and **rd16(in + off + 2)** before checking that two, let alone four, bytes remain: with **in_len** odd, the header read itself runs off the end. And a fourth thing that is not yet a bug: **off** and **in_len** are **int**, so a caller who computes **in_len** from a **size_t** has a truncation waiting, and **off += 4 + len** is a signed addition that could be made to overflow if **len** were not bounded — which it is, here, by accident of line (1). Note that repairing line (1) alone, as a reviewer in a hurry would, leaves the two serious bugs untouched. We audit the *argument*, not the line. ▲

3.4 Two case studies in the sixth shape

Sudo, CVE-2021-3156

Sudo's argument handling contained a loop that unescaped backslashes while copying user-supplied arguments into a buffer sized from an earlier pass. Under **sudoedit -s**, a mode combination reachable by any local user, an argument ending in a single trailing backslash caused the unescaping loop to consume the escape character and then read and copy the byte past the end of the argument, advancing the destination pointer past the allocation — a heap overflow with attacker-chosen contents and attacker-chosen length. The defect was introduced in 2011 and disclosed by Qualys in January 2021: nearly a decade in one of the most-audited setuid binaries on earth, in a default configuration, across every version from 1.8.2 to 1.8.31p2 and 1.9.0 to 1.9.5p1.

The bug is the sixth shape: a size decided in one pass and consumed by another, the two passes disagreeing about escape semantics. The reachability condition was a *mode combination* rather than an input format, and that is the kind of precondition a naive fuzzing harness never establishes; §5.3 is about exactly this.

But the part I want you to keep is the third one. Ubiquity and scrutiny are not the same thing, and the intuition that "surely someone has looked at **sudo**" is worth nothing. People had looked. They had not looked at that loop with that mode set.[3]

libwebp, CVE-2023-4863

WebP's lossless format encodes Huffman tables. **ReadHuffmanCodes** allocated the table buffer using a size drawn from a precomputed array, **kTableSize**, indexed by the colour-cache bits. That array accounted for the sizes required by eight-bit *first-level* lookups. But the format permits codes up to fifteen bits, and codes longer than the root size require *second-level* tables, whose space **kTableSize** did not include. Given a crafted image, **BuildHuffmanTable** filled second-level tables past the end of the allocation: a heap out-of-bounds write reachable by rendering an image. It was exploited in the wild before disclosure, and because libwebp is vendored nearly everywhere, the patch cascade ran through browsers, mail clients, chat applications, and image tools for months.

This is the sixth shape again, in its purest form: *the size was computed by one model of the format and consumed by another*. And note what would have found it. Not grep: there is no dangerous function call. Not a naive fuzzer, easily: reaching the second-level path requires a valid enough lossless stream with a sufficiently skewed code-length distribution, which random mutation reaches slowly. What finds it is a human asking the sixth question, *who decided this size, and does the consumer agree?*, or a fuzzer whose harness and dictionary were built by someone who had asked it.

3.5 Reading with a threat model

Concretely, this is what a first pass over an unfamiliar C codebase looks like. It takes a day for a small library and a fortnight for a large one, and skipping it is the most common reason a candidate's first engagement produces nothing.

1. **Enumerate the attack surface $\mathscr{S}$.** Every entry point that accepts bytes from outside $\mathscr{T}$: file formats, network parsers, IPC, environment, command line, and the ones people forget: filenames, metadata, error strings, and anything reached only after authentication that a low-privilege actor still reaches.
2. **Rank by depth and reachability.** Prefer surfaces that are unauthenticated, reached early, and parse a format with lengths in it. Deprioritise anything requiring privileges the attacker would not have.
3. **Locate the sinks.** Memory: **memcpy**, **alloca**, indexed writes, **malloc** arithmetic, **free**, casts and unions. Non-memory: **system**, **exec**, path construction, SQL, deserialisation, format strings.
4. **Walk backwards from each sink to a source**, writing down for every operand the guard that establishes its precondition, or the word **NONE**. The document you are producing is the audit. If you are not producing a document, you are browsing.
5. **Attack the guards you found** with the three failure modes from Theorem 3.2: wrong variable, wrong time, wrong type.

[3] The mode combination is **sudoedit -s**. I have watched people read the advisory, nod, and then not go and check whether their own targets have an equivalent second mode. There is usually a second mode.

PLATE 3 Every window on this elevation was made to one drawing, by one trade, at one time. If a single unit was hung wrong, the useful question is not whether *that* window can be fixed but how many others share the defect. That is knowable in advance, because they share a cause rather than merely a resemblance. Variant analysis is this observation applied to code. The second instance costs a fraction of the first; the eleventh is nearly free.

6. **Diff against history.** Read the last two years of security-relevant commits. Maintainers fix one instance of a mistake they make habitually; §3.6 is how you collect the rest.

3.6 Variant analysis, and how one bug becomes eleven

The most consistently underrated technique in the field is harvesting the bug classes already found. *Variant analysis* takes a known defect, abstracts it into a pattern, and sweeps the pattern across the codebase, and then across every codebase you can index.

The method does not require a query language to begin. *One*, understand the original precisely enough to state it as a rule. *Two*, write a pattern that matches exactly the known instance; start over-specific, so you can verify the pattern is right. *Three*, identify what may be generalised: the function name, the type, the guard's shape, the direction of the comparison. *Four*, generalise one element at a time, re-running after each step and reading the new matches. *Five*, triage.

Tooling scales with ambition: **ripgrep** for the first hour and further than candidates expect; **semgrep** for syntactic patterns with metavariables; **CodeQL** when you need dataflow, taint, or path conditions, and when you want to run the same query over a thousand repositories at once. GitHub's Security Lab has published a long series of write-ups in which a single seeded vulnerability, generalised into a query, yields a family. Their Ghostscript type-confusion work is the classic teaching example, where a query modelled on a handful of prior bugs surfaced several fresh ones.

The habit to build is this: *never report one bug*. When you find something, spend the next two hours asking what class it belongs to, whether the author repeated it, and whether the pattern would match elsewhere. The second bug costs a fraction of the first, and the eleventh is nearly free.

RANGE SAFETY

Static analysis output is not a finding. A tool that reports 4,000 hits on a codebase has told you nothing until you have established reachability for a specific hit, and pasting a tool's output into a report — or into a maintainer's issue tracker — is the fastest way to be permanently ignored by a project you may want to work with for years. The rule at this Institute is that no finding leaves the building without a candidate having personally constructed the input that reaches it, or having written, in the report, the exact sentence explaining why they could not and what that leaves uncertain.

NOTES AND SOURCES

The reading method of §3.5 — attack-surface enumeration, trust-boundary analysis, backward walks from sinks — is substantially that of [16], and the debt is larger than a citation conveys. The source, sink and guard vocabulary is standard in taint analysis and matches what CodeQL and Semgrep call the same things. The three failure modes in the proof map onto documented weakness classes: checking the wrong variable, checking at the wrong time, and checking in the wrong type [11].

Theorem 3.2 was called "Prather's theorem" in earlier printings and dated to 2003. D. Prather is a character in this book's framing and the date was invented; both have been withdrawn, and the front matter lists the rest. Nothing about the argument changes. The forward direction is ordinary path induction over the control-flow graph, of the kind any treatment of dataflow soundness will give you, and the converse restates what taint analysis means by a sanitiser.

On **calloc**, overflow checking is required by CERT MEM07-C [11] and is universal in mainstream implementations, but it is not compelled by the standard itself and historically was not guaranteed.

Every case in §3.3 is a published defect with a public root-cause account, and the accounts are better than my summaries of them. Shape (i) is Stagefright [17]; (ii) is Sequoia [42]; (iii) is wget [54]; (iv) is Lhasa [18]; (v) is Netatalk [31] and GHOST [41]. Caveats about how they are used here. The pairing of one case to one shape is a teaching decision and not a claim about taxonomy: wget is filed under signedness but narrows into an **int** as well, and GHOST is an omitted term in a size sum rather than a terminator error strictly construed. That is why the section says so in the text rather than quietly filing it. And the "what finds it" notes are the author's judgement about method, informed by how each defect was in fact found. No detection rate was measured. Where a real detection history exists it is stated: Stagefright was found by an AFL campaign, and the Sequoia trigger needs resources no ordinary campaign brings.

The sudo case is Qualys' [40], which gives the function, the loop, the mode combination and the introducing commit. The libwebp case is CVE-2023-4863 [29]; the out-of-bounds write occurs in **ReplicateValue**, at a remove from where the size was decided. That is §8.3's lesson in miniature. The variant-analysis example is GitHub Security Lab's Ghostscript work [20]: a query modelled on prior type confusions yielded six new ones from fifteen results after filtering — a published instance of the precision curve Exercise 3-5 asks you to plot.

Exercises

Starred exercises require a licensed range, a written scope, and a supervising Second Bench. Unstarred exercises require a text editor. Do not confuse the two.

Exercise 3-1. Repair the parser of Worked Example 3-1 completely. Then write, in two sentences each, the guard that each of your changes establishes and the precondition it establishes it for. If you cannot name the precondition, you have written a patch, not a fix.

Exercise 3-2. For each of the six arithmetic shapes in §3.3, write a minimal C function exhibiting it. Then build every one of them four ways: under ASan, under UBSan with the default check set, under UBSan with **unsigned-integer-overflow** added, and under MSan. Record which configuration reports the misbehaviour, and on which inputs. Two of the six are reported by none of the four. Say which, and say why that answer is a property of the C standard rather than of the tools.

Exercise 3-3. Take the libwebp case. Without reading the patch, write down the invariant that **kTableSize** was supposed to guarantee, and the assertion you would add to the allocation site to make its violation loud. Then read the patch and compare.

Exercise 3-4. You are handed a 400,000-line C codebase and one week. Write the one-page plan. It must name the surfaces you will *not* look at and justify each exclusion, because a plan that does not exclude anything is not a plan and will produce a report that says "time did not permit".

Exercise 3-5. * (Supervised; own code or an authorised target.) Pick a public, fixed CVE in a C project. Write a Semgrep or CodeQL query that matches the pre-patch code and not the post-patch code. Then generalise it in three steps as in §3.6, running the query over the project at each step, and record the count of matches and the count of true positives after each generalisation. Plot the two curves. The shape of the gap between them is the thing this exercise teaches, and no amount of reading about precision and recall substitutes for having drawn it once yourself.

CHAPTER 4

Dynamics: Fuzzing

The fuzzer did not get lucky. The fuzzer got told.

— Y. Sorokin, marginal note in the first edition of this book, in pencil, with two exclamation marks I have chosen not to reproduce.

BY THE END OF THIS CHAPTER

Explain why coverage feedback converts a product of constraint probabilities into a sum, and use that to predict where a campaign will stall. Budget a campaign from an estimate of the trigger probability, and, running the arithmetic backwards, state the bound a campaign that found nothing has established. Diagnose a plateau instead of waiting it out. Never report a crash count as though it were a bug count.

4.1 From reading to running

Chapter 3's method answers CONTROL superbly and REACHABILITY badly. You can read a guard and know exactly what it fails to establish; you cannot read four hundred thousand lines and know which of them a network packet reaches. Fuzzing is the complementary instrument: it answers REACHABILITY empirically, at millions of samples per hour, and it answers it about the program that actually built rather than the program you believe you read.

The naive version is Miller's and it still works: generate random bytes, feed them in, watch for crashes. In 1990 that crashed somewhere between a quarter and a third of the Unix utilities he pointed it at. Try the same thing today against a parser anyone has cared about and you will get nothing, and you will get nothing for as long as you care to run it. Why that changed is more or less the whole of this chapter.

4.2 The arithmetic of the constraint chain

Consider a bug reachable only when the input satisfies a chain of conditions: a magic number, a version field, a type tag, a length in a particular range, a flag set. Model each as an independent equality on d bits. To reach the bug blindly, all k must hold at once:

$$p_{\text{blind}} = \prod_{i=1}^{k} 2^{-d_i} = 2^{-\sum_i d_i}. \tag{4.1}$$

For a four-byte magic, a one-byte version, and a two-byte tag (a modest, entirely ordinary file header) that is 2^{-56}. At 10^5 executions per second you will wait about twenty-three thousand years.[1]

Now instrument the target so the fuzzer learns which *edges* of the control-flow graph an input executed, and keep any input that reaches an edge no earlier input reached. The magic number check, once passed

[1] Rounded. The exact figure is not more useful than the rounded one and I have never once needed it.

by any input, is passed by every descendant of that input, because the fuzzer saved it and mutates from it. The product collapses:

$$W_{\text{blind}} = \prod_{i=1}^{k} 2^{d_i} \qquad \longrightarrow \qquad W_{\text{guided}} \approx \sum_{i=1}^{k} 2^{d_i}. \tag{4.2}$$

Principle 4.1 (Why coverage feedback works) *Coverage-guided fuzzing converts a* product *of constraint probabilities into a* sum*. Every saved input is a solved prefix that never has to be solved again.*

That single sentence is the most important idea in this chapter, and everything practical follows from it as engineering. 2^{56} becomes $2^{32} + 2^{8} + 2^{16}$, which at 10^5 executions per second is about twelve hours instead of twenty-three thousand years; and then the techniques of §4.4 take the remaining 2^{32} apart as well.

What gets turned around, in roughly the order it costs people time:

- **Throughput and feedback are not interchangeable.** Doubling r halves the time; adding feedback changes the exponent. A slow fuzzer with good feedback beats a fast one without, by margins that are not close. Optimise the exponent first.
- **Coverage is a proxy.** The fuzzer maximises coverage because coverage is a computable stand-in for progress. It is a good proxy and a bad target: a corpus can gain edges while getting worse at finding bugs, and §9.3 is about the fact that we still cannot measure the difference well.
- **The constraint chain belongs to your harness.** It is not a fixed property of the program. A harness that starts *after* the magic-number check deletes 2^{32} from the exponent for free. That is why Chapter 5 argues that harness design is where the skill lives.

4.3 How long should this take?

Once a bug's guarding constraints are solved, triggering it has some per-execution probability p. Then over N executions

$$\Pr[\text{found}] = 1 - (1-p)^N \approx 1 - e^{-pN}, \qquad \tau = \mathbb{E}[\text{time}] = \frac{1}{pr}, \tag{4.3}$$

where r is executions per second. You can run it either way. *Forwards*: if you believe $p \approx 10^{-7}$ and you are running at $r = 2 \times 10^4$, then $\tau \approx 500$ seconds and a campaign that has run for a day without hitting it has told you your estimate of p was wrong by orders of magnitude. That is information. *Backwards*: a campaign that has found nothing for a week bounds p from above, and that bound, stated in a report, is worth far more than the sentence "we fuzzed it".

WORKED EXAMPLE 4-1 Budgeting a campaign

A library parses a container format. Your harness runs at $r = 8 \times 10^3$ exec/s with ASan on one core; you have 16 cores and three days. Total executions $N = 8 \times 10^3 \times 16 \times 2.6 \times 10^5 \approx 3.3 \times 10^{10}$. For a bug with $p = 10^{-9}$, $\Pr[\text{found}] = 1 - e^{-33} \approx 1$: comfortably inside budget. For $p = 10^{-12}$, $\Pr \approx 3\%$: outside it, and no amount of staring at the coverage graph will change that. What changes it is a better harness (raising p by removing constraints) or a dictionary (raising p by supplying the tokens). Note the asymmetry that governs practice: cores buy you a linear

factor, and harness work buys you an exponent. Which is why the request that arrives on my desk is almost always for cores, and almost never for a week to rewrite the harness. Cores are a purchase order. The harness is a fortnight of somebody's attention, and there is no form for that. ▲

4.4 Making the mutations count

Given the chain-collapse principle, everything else in fuzzing is a technique for solving one link more cheaply. The important ones, in the order you should reach for them:

Seeds. Start from real, valid, *small*, *diverse* inputs. A good corpus solves most of the chain before the fuzzer runs at all. Small matters more than candidates expect: mutation energy is spread over the input, so a 4 KB seed explores far better than the 2 MB one you found first.[2] Gather seeds from the project's own test suite, from format conformance suites, and from public corpora, then minimise the set (§4.5).

Dictionaries. A list of format tokens (magic bytes, keywords, tag names) that the mutator can splice in whole. This directly supplies the constant side of an equality constraint. For any format with textual keywords the dictionary is the single highest-return hour you will spend.

Comparison unrolling and input-to-state. Modern engines instrument comparisons and feed the operands back into mutation. AFL++'s CmpLog (following the Redqueen work) records the values a comparison actually saw and tries substituting them into the input at matching offsets; LAF-Intel/**-fsanitize-coverage=trace-cmp** splits multi-byte comparisons into byte-wise ones so that coverage feedback can climb them one byte at a time. Between them these turn a 32-bit magic check from 2^{32} into something close to free, and they are the reason a modern fuzzer solves headers a 2013 fuzzer could not. Turn them on. In AFL++ this means building a second, CmpLog-instrumented binary and passing it with **-c**.

Structure-aware mutation. When the format has checksums, lengths that must agree with content, or compressed sub-streams, byte-level mutation is fighting the format rather than the program. Either write a custom mutator that understands the format (libFuzzer's **LLVMFuzzerCustomMutator**, AFL++'s custom mutator API), or, usually better, change the *input language*: define the input as a protocol buffer, let **libprotobuf-mutator** mutate the message, and have the harness serialise it into the real format, computing lengths and checksums itself. The fuzzer then cannot generate an invalid input, and every execution is spent on program logic instead of on CRC roulette.

Ensemble. No engine dominates, and running several is cheap and reliably beats any single configuration.[3] Run AFL++, libFuzzer or Centipede, and honggfuzz against the same corpus directory and let them share finds; run several AFL++ instances with different power schedules (**-p explore**, **fast**, **seek**, **rare**) and different instrumentation.

4.5 Corpus and sanitiser hygiene

What separates campaigns that keep improving from campaigns that plateau in a day is a pair of dull habits.

[2] I have never seen anyone regret minimising a corpus. I have twice seen someone minimise the wrong directory.

[3] Cheap in cores. Not cheap in the half-day you will spend discovering that two of the engines disagree about what a corpus directory is.

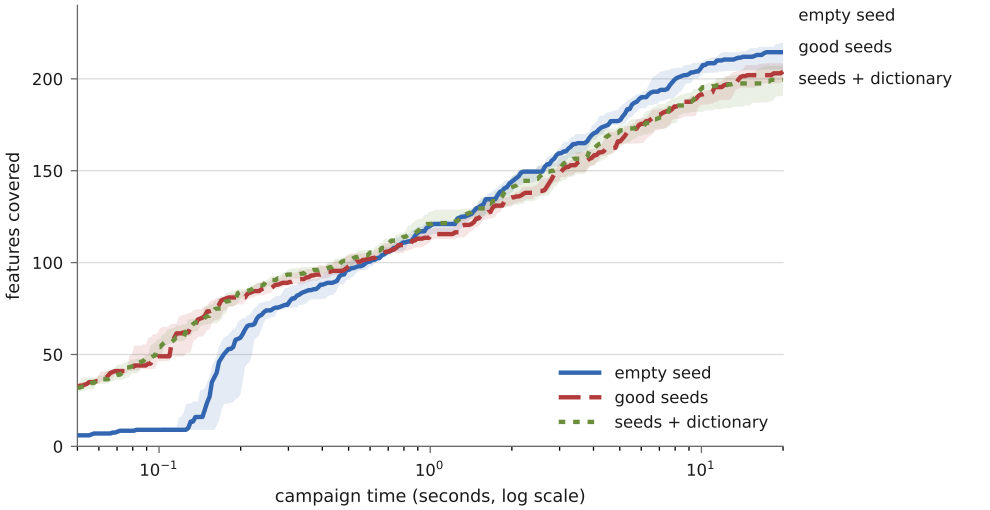


Figure 4-1 Coverage against log time for one target under three configurations: median of ten trials each, interquartile band shaded. Read what is robust in this plot separately from what is not. The *flattening* is robust — every configuration decelerates, and no amount of further time recovers the early slope. The *early advantage of seeding* is robust: at fifty milliseconds the seeded runs have five times the coverage of the empty-seed run, because a seed is a solved constraint chain you did not have to solve. What is *not* robust is the ordering at the end. Here the empty-seed campaign overtakes both seeded ones after about a second and finishes with the most features. The reason is §4.4: this target's gates are byte comparisons against constants, exactly what a modern engine's input-to-state substitution solves for free, so the seeds buy a head start and then constrain the corpus to their own shape. On a format the engine cannot solve that way — a checksum, a compressed sub-stream — the ordering returns. Measure your own target; do not inherit a curve from a book.

Minimise the corpus, repeatedly. **afl-cmin** (or **libFuzzer -merge=1**) selects the smallest subset of inputs preserving coverage; **afl-tmin** shrinks each input. A corpus that has grown to forty thousand files is mostly spending your executions on redundancy. Re-minimise weekly.

Fuzz more than one build. The standard rotation is: a plain build for maximum throughput, an ASan+UBSan build for oracle strength, a CmpLog build for constraint solving, and, if you can afford to stand it up, an MSan build, because uninitialised-memory bugs are invisible to everything else and are over-represented in information-disclosure findings. Give each a share of your cores and sync their corpora. If you run only one build, run the sanitised one: silent corruption is worse than slow.

4.6 The plateau, and what to do about it

Every campaign flattens. The flattening is not failure; it is the fuzzer telling you it has exhausted what your current configuration can reach, and the correct response is diagnostic rather than patient.

Work down this list in order. *Is the coverage where you think it is?* Generate a coverage report (**llvm-**

cov over the corpus) and *look at it*. Candidates who do this for the first time routinely discover that the campaign never entered the subsystem they cared about, usually because of an early **return** on an unmet precondition. *Is a checksum eating you?* Patch it out in the fuzz build. A build flag that stubs verification is standard practice, and is why nearly every OSS-Fuzz integration carries a small patch. *Is the harness too narrow?* Add entry points; fuzz the encoder as well as the decoder, an asymmetry that leaves real bugs sitting in mature projects for years because researchers instinctively attack the decode path. *Is the input language wrong?* Move to structure-aware generation. *Is the oracle too weak?* Add a sanitiser, or add assertions: a well-placed **assert** on an internal invariant converts silent wrongness into a crash, and invariant-hunting is one of the highest-yield collaborations you can have with a maintainer.

ON THE WORD "UNIQUE"

Your fuzzer will report a number of "unique crashes". That number is not the number of bugs, is not proportional to the number of bugs, and should never appear in a report. AFL-style coverage-based uniqueness can inflate a single bug into hundreds of entries; stack hashing, the usual alternative, does better but still over-counts, and in the most careful study of the question a single bug mapped to dozens of distinct stack hashes while roughly a sixth of hashes were shared between *different* bugs, so it under-counts too, silently discarding real findings. The operational rule is simple: crashes are raw material, bugs are a conclusion you reach in Chapter 8 by minimising and root-causing, and the only honest sentence in the interim is "*n* crashing inputs, not yet triaged". Anyone who tells you their fuzzer found four hundred vulnerabilities has told you they have not looked at them.

RANGE SAFETY

Four ways a fuzzing campaign has hurt someone at this Institute, all real, all preventable. *Disk*: an untended campaign wrote 900 GB of crash inputs to a shared filesystem overnight; set **-M** limits, put output on local storage, and monitor. *Network*: a harness for a client library resolved and connected to hostnames drawn from mutated input, and the Institute spent a fortnight explaining itself to three ISPs. Fuzz with the network disabled — a namespace, a seccomp filter, a stubbed resolver — unless you have written permission naming the hosts. *Filesystem*: a target that creates files from input-derived paths will eventually create **../../something**; run it in a container with nothing mounted that you would miss. *Other people's machines*: fuzzing a service you do not own is an unauthorised availability attack whatever your intentions, and the statutes turn on authorisation. If it is not yours and you do not hold a signed scope, do not point a fuzzer at it.

NOTES AND SOURCES

The blind-versus-guided argument is the standard justification for coverage-guided fuzzing, and its practical consequences are visible throughout [2]. Comparison unrolling and input-to-state substitution come from Redqueen [5] and are implemented as CmpLog in AFL++ [2]. Note that splitting multi-byte comparisons into byte-wise ones and reporting comparison operands back to the runtime are two different mechanisms with related effects; earlier printings of this book conflated them.

The de-duplication figures in the sidebar are from [28]: a single bug mapped to about forty-six stack hashes on average, roughly 16% of hashes were shared between genuinely different bugs, and in five cases a distinct bug's only crash carried a non-unique hash and would have been discarded outright. The same paper found that seventeen of thirty-two surveyed fuzzing papers report no trial count at all. Its authors ran at least thirty trials per configuration; the ten asked for in Exercise 4-6 are a concession to a term's timetable and are not a standard. The statistical machinery that exercise names is from [4] and [52].

Figure 4-1 is measured rather than modelled: thirty campaigns, ten per configuration, against the target and harness in the accompanying materials. Its partial disagreement with the received picture is discussed in the caption, and is worth taking seriously as a caution about inheriting curves from books, including this one.

Exercises

Starred exercises require a licensed range, a written scope, and a supervising Second Bench. Unstarred exercises require a text editor. Do not confuse the two.

Exercise 4-1. A parser checks a 4-byte magic, then a 1-byte version, then a 2-byte record count that must lie in $[1, 8]$. Compute p_{blind} and the expected blind time at 5×10^4 exec/s. Then compute the expected *guided* work under Principle 4.1, and state which single one of the three constraints dominates it and why.

Exercise 4-2. Your campaign runs at 2×10^4 exec/s on 8 cores and has found nothing in 48 hours. Derive the resulting upper bound on p for any bug reachable by your harness. Then write the two sentences you would put in a report: one stating what this establishes, one stating what it does not.

Exercise 4-3. Explain, with reference to Principle 4.1, why adding a dictionary usually helps more than doubling the core count, and construct a case where the opposite is true.

Exercise 4-4. You are told a target's coverage has plateaued at 34% of lines after a week. List, in the order you would try them, six actions, and for each of them the observation that would tell you whether it worked. "Run it longer" may appear on your list only with a justification.

Exercise 4-5. A colleague proposes reporting "1,200 unique crashes". Rewrite the claim honestly in one sentence, and describe the smallest amount of work that would let you replace it with a number that means something.

Exercise 4-6. * (Supervised; own code.) Write a libFuzzer harness for a small parser you control. Run it four ways (empty seed; seeded; seeded plus dictionary; seeded plus dictionary plus CmpLog) for one hour each, ten trials each, recording edges covered and time to first crash. Compare the configurations with a Mann–Whitney U test and report the Vargha–Delaney $\hat{A}_{12}$ effect size. Ten trials is the minimum any claim about fuzzer performance is entitled to; the field's own methodological reckoning found that most published comparisons used fewer trials than this exercise requires of you in week six, and you should let that inform how you read fuzzing papers for the rest of your career.

CHAPTER 5

Topology: Coverage, Harnesses, and Reach

Strength is a number. Reach is a yes or a no.

— Y. Sorokin, *Reachability and Residue* (2011), first sentence.

BY THE END OF THIS CHAPTER

State the coverage bound and its consequence: that a campaign's findings are a property of what it executed, so a report without a coverage figure is not a report, and an instrument that over-approximates reach is not interchangeable with one that under-approximates it. Then write down a caller's contract before you write the harness, classify the kind of surface you are cutting into, and recognise the two symmetric failures: reaching too little, and reaching what no attacker can.

5.1 The coverage theorem, and what you cannot find

Everything in Chapter 4 was about spending executions well. This chapter is about the prior question: which parts of the program those executions can touch at all. It has a one-line answer, and the answer is senior to every technique in this book:

Theorem 5.1 (Coverage bound) *No dynamic technique can report a defect in code it does not execute. If E is the set of edges executed over a campaign, the campaign's findings are a subset of the defects resident in E, regardless of duration, engine, oracle, or core count.*

Proof. Immediate: a dynamic oracle is a predicate on observed states, and unexecuted code contributes none. □ □

The theorem is trivial and its consequences are not. "We fuzzed it for a month and found nothing" is a statement about E and not about the program, and a campaign report which does not include a coverage figure is not a report. The question you must be able to answer about any engagement, and the question a good reviewer will ask first, is not *how long did you run* but *what did you reach, and what did you not, and why*.

The practical form of that question is half an hour with **llvm-cov**, reading the annotated source and finding the early **return** that has been silently discarding 80% of your inputs since Tuesday.[1]

[1] It is always Tuesday.

PLATE 4 The arch stood there all night. The beam does not create it; it bounds what can be seen, and nothing outside the lit cone can be reported on, not because it is absent but because it was never illuminated. Theorem 5.1 says precisely this about a campaign, and it is why "we fuzzed it for a month and found nothing" describes the beam rather than the program.

5.2 What kind of surface is your target?

Before writing a harness, classify the target, because the classification determines the whole approach.

Stateless transformers. One input, one output, no memory between calls: image decoders, parsers, decompressors, regex compilers. These are the easy case and the reason the fuzzing literature is full of image decoders. A single **LLVMFuzzerTestOneInput** does the job.

Stateful APIs. The behaviour depends on a sequence of calls: an allocator, a database handle, a TLS connection, a video decoder with reference frames. Feeding raw bytes to one function tests almost nothing, because the interesting bugs live in the transitions. The technique is to treat the input as a *program*: interpret the fuzzer's bytes as a sequence of opcodes with arguments, and execute the sequence against the API. Either hand-roll this with a **FuzzedDataProvider**, or, for anything beyond a handful of operations, define the sequence as a protobuf message and let a structure-aware mutator manipulate it.

Stateful protocols. As above, plus the peer is remote and there is a handshake, a sequence number, and probably cryptography. Here the harness work dwarfs the fuzzing work, and the standard move is to fuzz the protocol *library* in-process rather than the service over a socket: two orders of magnitude more executions per second and vastly better reproducibility.

Kernels and privileged interfaces. A special case of stateful, with resources (file descriptors, handles) produced by one call and consumed by another, and where a crash takes the machine with it. This is what **syzkaller** exists for: system calls are described declaratively in **syzlang**, including which calls produce and consume which resource types, so the generator emits sequences that are structurally valid; coverage comes from **kcov**, the oracle from KASAN and friends, and the whole thing runs inside disposable virtual

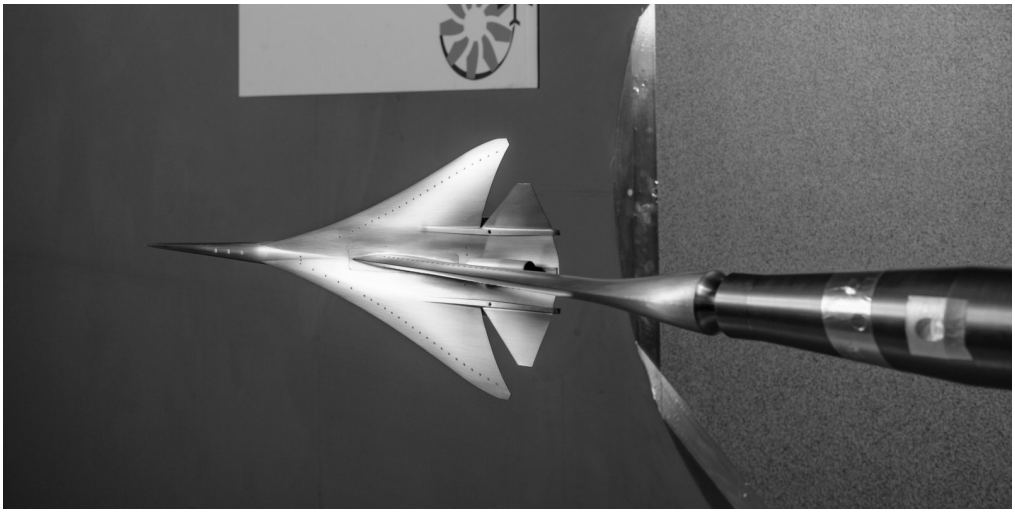


PLATE 5 A scale model on a sting mount in a supersonic wind tunnel. The model is not the aircraft: it is smaller, it is not flying, the air is moved past it rather than the reverse, and it is held from behind by a strut no aircraft carries. Every one of those departures is deliberate, and every one is a threat to the result. This is the harness problem in metal. Build the rig too faithful to the original and it teaches you nothing the original could not; build it too loose and it teaches you about the rig. *NASA.*

machines. Nothing in this paragraph is optional if you intend to fuzz a kernel, and §7.6 assumes it.[2]

5.3 Doors: the harness, and why it is the whole job

Definition 5.2 (Harness) *A harness $\mathscr{H}$ is a program that maps a byte string to a sequence of calls into the target, together with the build configuration under which those calls run. Its* fidelity ϕ *is the degree to which the states $\mathscr{H}$ can construct correspond to states reachable by a real attacker across $\mathscr{T}$.*

A harness is a door you cut in the program's surface. Cutting one is necessary, since there is no fuzzing without it, and cutting it badly is worse than not cutting it at all, because a bad harness produces confident wrong answers in both directions.

Fidelity too high (under-reaching). The harness faithfully reproduces the real entry point, including a validation layer, an authentication step, or an initialisation that constrains the input. The fuzzer spends its life in the front door and never reaches the interesting code. Symptom: coverage plateaus early and the covered set is all in one file.

Fidelity too low (over-reaching). The harness calls an internal function directly, skipping invariants that every real caller establishes, and this wastes more candidate-weeks than any other single mistake in this book. Now the fuzzer finds crashes constantly, and they are all harness artefacts, states no attacker can

[2]People do try to write their own. I have seen it attempted three times and finished zero times, which is not an argument, only an observation with a small sample.

produce. Symptom: a flood of crashes in the first minute, all in the same function, none of which survive Chapter 8's reachability argument. The first ten of those minutes feel wonderful.

The discipline that prevents both is to write down, before the harness, the *contract*: what does every legitimate caller of this function guarantee about its arguments? Then make the harness establish exactly those guarantees and nothing more. Where you deliberately relax one, stubbing a checksum say, record it, because it becomes a caveat in the report and possibly the reason a finding is rejected.

The rules, each of them load-bearing:

1. **Deterministic.** No time, no randomness, no threads unless the bug is a race, no network, no persistent global state between iterations. A non-deterministic harness makes minimisation and root-causing (Chapter 8) nearly impossible, and a crash you cannot reproduce is not a finding.
2. **Fast and small.** Cap input size in the harness. Skip expensive initialisation by hoisting it out of the loop: libFuzzer's static-initialiser idiom, AFL++'s deferred forkserver and persistent mode. Every millisecond in the harness is executions you do not get.
3. **Leak-clean.** Free what you allocate, or LeakSanitizer will bury you in harness-caused reports. This is boring and it is the difference between a usable campaign and a noisy one.

WORKED EXAMPLE 5-1 A harness, wrong and then right

Suppose **img_decode(const uint8_t *buf, size_t n, img *out)** requires that **out** has been initialised by **img_init** and that **n <= MAX_IMG**. The instinctive harness is

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *d, size_t n) {
    img out;                        /* uninitialised: contract broken */
    img_decode(d, n, &out);         /* n unbounded: contract broken */
    return 0;                       /* out never released: leaks */
}
```

Every finding this produces will be worthless, and — worse — your oracle decides which ones you even see. Under the ASan build this book recommends everywhere else, the uninitialised read of **out.stride** is *silent*: **img_decode** consumes stack garbage, returns a plausible result, and nothing fires. Only the leak is reported, and only at exit. To see the read at all you need MemorySanitizer, for the reason given in §2.5: ASan does not find uninitialised memory. A harness that breaks its contract in three ways can look almost healthy under the wrong instrument. The honest version establishes the contract, bounds the input, and cleans up:

```c
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *d, size_t n) {
    if (n > MAX_IMG) return 0;      /* mirror the caller's bound */
    img out;
    img_init(&out);                 /* establish the precondition */
    if (img_decode(d, n, &out) == 0)
        img_consume(&out);          /* exercise the success path too */
    img_free(&out);                 /* no leak noise */
    return 0;
}
```

Note the third line from the bottom. A harness that decodes and discards tests the decoder; a harness that decodes and then *uses* the result tests the decoder and the contract between decoder and consumer, and the second is where the type confusions live. Most harnesses in the wild stop at decode, and that is a large part of why bugs survive years of continuous fuzzing in projects that are, on paper, thoroughly fuzzed. ▲

5.4 When the fuzzed volume is correctness

Nothing in Definition 5.2 requires the oracle to be a crash. *Differential* fuzzing runs two implementations of the same specification on the same input and compares their outputs, so the fuzzer becomes a search for *disagreements* rather than for corruption, and it reaches an entire category that Chapters 3 and 4 cannot see.

The category is the one from §1.1: two components with incompatible beliefs. Two X.509 parsers that disagree about a certificate's subject; a firewall and a server that disagree about where an HTTP request ends; a signature verifier and a payload extractor that disagree about which bytes were signed; two JSON parsers that disagree about duplicate keys. None of these crashes. All of them are vulnerabilities, and several of the most consequential findings of the last decade are of exactly this shape.

The mechanics are pleasant: the harness runs both implementations, normalises the outputs, and calls **abort()** on mismatch, at which point every technique in this book applies unchanged. The hard part is choosing what to compare and deciding which disagreements are security-relevant, which returns you to the threat model of §3.5 and to thinking.

A related inward-facing trick: fuzz your target against *itself*. Encode then decode and require the round trip to be the identity; compress then decompress; serialise then parse. Round-trip oracles are cheap to write, need no second implementation, and are extremely good at finding the encoder-side bugs that §4.6 noted everyone neglects.

5.5 Links: dataflow, taint, and feasible paths

The last topological idea is the one that connects Chapters 3 and 4. Whether a source and a sink are joined by a *feasible* path is not a matter of code proximity; it is a matter of dataflow, and dataflow is invariant under most of the things that make code look different. Write $L(s,k)$ for the existence of a feasible path from source s to sink k. L is what a taint-tracking query computes, what a symbolic executor tries to witness, and what a fuzzer establishes by construction when it produces an input.

The instruments, with honest accounts of their limits:

Taint and dataflow queries (CodeQL, and the whole family). These compute an *over*-approximation of L: they will show you paths that no input can actually take, because they abstract away the path conditions. That is the correct trade for variant analysis, where you want recall and you triage, but it means a query hit is a hypothesis rather than a finding.

Symbolic and concolic execution (KLEE, angr, SymCC, and hybrid schemes such as Driller). These compute an *under*-approximation: what they return is a real input for a real path, because a solver constructed it. The limit is path explosion, branches multiplying states exponentially, and the practical use is therefore surgical rather than global: point a concolic engine at the one comparison your fuzzer cannot get past, take the input it produces, put it in the corpus, and let the fuzzer carry on. Hybrid fuzzing is essentially this loop, and the modern engines' input-to-state substitution (§4.4) has absorbed much of what it used to be needed for.

The fuzzer itself, which computes L constructively and is wrong only in the safe direction: what it produces is real. This is why, when a static query and a fuzzing campaign disagree about whether something is reachable, you should believe the fuzzer when it says *yes* and believe nobody when it says *no*.

RANGE SAFETY

Symbolic execution and heavy instrumentation are resource events. A single **angr** exploration has consumed 200 GB of resident memory on a shared machine at this Institute and taken the machine's other eleven users with it. Set memory limits at the process level — not in your own code, where the allocator failure will simply be a crash you then spend a day triaging. And note the specific trap: an out-of-memory kill and a memory-corruption abort are indistinguishable in a poorly configured campaign, so a fuzzer that is mysteriously "finding" crashes on large inputs is usually finding your **-rss_limit**.

NOTES AND SOURCES

The harness rules of §5.3 restate libFuzzer's own guidance [14]. The kernel arrangement — declarative system-call descriptions with resource types, coverage from the kernel's own instrumentation, kernel sanitisers as the oracle, and disposable virtual machines — is syzkaller's [50].

Every example in §5.4 is a real case and repays reading in full. Disagreeing certificate validators: frankencerts [9], where eight million synthetic certificates and a quarter of a million real ones produced 208 discrepancies and serious flaws in four independent TLS implementations. Disagreement about where a request ends: HTTP request smuggling [27], a class first published in 2005, rediscovered in 2019, and still yielding in 2025. Disagreement about which bytes were signed: Android Janus, CVE-2017-13156, where one file is simultaneously a valid archive and a valid executable because the signature scheme considers only the archive entries while the loader reads the header. Disagreeing JSON parsers: forty-nine implementations surveyed [6], and Seriot's [46] finding that no two behaved alike. Note that three of those four are not memory-safety bugs.

On the instruments, taint queries over-approximate and symbolic execution under-approximates [10, 47, 39]; hybrid schemes [49] combine them, though much of what hybrid fuzzing was needed for has since

been absorbed by input-to-state substitution [5].

Exercises

Starred exercises require a licensed range, a written scope, and a supervising Second Bench. Unstarred exercises require a text editor. Do not confuse the two.

Exercise 5-1. Your campaign covers 41% of lines. Write down five distinct reasons the other 59% might be unreached, ordered by how cheaply you could distinguish them, and for each the observation that distinguishes it.

Exercise 5-2. Classify each target and state the harness strategy in one sentence: (a) a PNG decoder; (b) a TLS record layer; (c) an allocator; (d) an ELF parser; (e) an ioctl handler in a kernel driver; (f) a URL normaliser used by both a cache and an origin server.

Exercise 5-3. Take the wrong harness of Worked Example 5-1. For each of the three defects, describe the specific false finding it produces and how you would recognise, from the ASan report alone, that the finding is a harness artefact rather than a bug.

Exercise 5-4. Design a differential harness for two implementations of the same archive format. Specify precisely what you compare (not "the output"), how you normalise, and three classes of disagreement you would deliberately *ignore*. The ignore list is the exercise; a differential harness without one reports nothing but noise.

Exercise 5-5. Explain why a static dataflow query and a symbolic executor make errors in opposite directions, and describe an engagement in which you would deliberately use both.

Exercise 5-6. * (Supervised; own code.) Instrument a target you control so that a checksum check can be disabled by a build flag. Run identical one-hour campaigns with the check on and off, ten trials each, and report edge coverage for both. Then write the two-sentence caveat you would place in a report to describe what the checksum-disabled campaign does and does not establish about the shipped binary. The caveat is what a reviewer will read.

CHAPTER 6

The Researcher: Yield, Match, and Limit

The campaign is only ever as good as the person who reads the coverage report.

— Assurance proverb, all campuses, no attested origin, no attested exception.

BY THE END OF THIS CHAPTER

Reconnoitre a target before spending effort on it, and estimate whether your method matches what is there. Recognise over-matching as a real and expensive error. Scope an engagement by choosing what you will not examine and writing that down. Plan work against sustainable rather than peak capacity. And know which parts of a search can be substituted by automation and which part — noticing that the space is wrong — cannot.

6.1 The researcher as a component

Every model so far has treated the searcher as given — an unmodelled source of harnesses, queries, and attention. This chapter is about that source. A vulnerability-research programme is executed by people, and a person is, in the engineering sense, a component: with a throughput, a matching characteristic, a duty cycle, and a failure mode. The purpose of this chapter is to give you the component model of yourself.

A warning about where that leads, so it does not read later as a swerve. Take the component model seriously and it stops being about throughput almost immediately, because the properties of this component that turn out to matter are its matching characteristic and its duty cycle rather than its speed. Those are the model's own conclusions, arrived at by taking it literally. By §6.6 the chapter is discussing rest, scope and what you decline to look at, and a reader who came for engineering may feel the floor move.

The alternative is to learn your own ratings the way my generation did: empirically, in the field, three days into a deadline. More people leave this profession that way than leave because the work got too hard. I can name six. Two of them were better at this than I am.[1]

6.2 Effort, match, and the yield theorem

Model an engagement as follows. Let E be effort — hours, cores, attention, whatever you are actually spending. Let the target have a *bug profile*: a distribution over where its defects live and what would surface them. Let your approach have a *method profile*: what your harnesses reach, what your queries match, what your reading habits notice. Write m for the ratio between them — how well what you are doing matches what is there. Then yield behaves, to a first approximation, like

$$Y \;=\; Y_{\max} \cdot \frac{4m}{(1+m)^2}, \tag{6.1}$$

maximised at $m = 1$ and falling away on *both* sides. Both sides: that is the sentence to keep.

[1] One of them now keeps bees. I mention it because candidates always assume the alternative to this work is a worse job, and it frequently is not.

EDITORIAL NOTE

(6.1) is an analogy, not a measurement. Its functional form is borrowed from the maximum-power-transfer relation in electrical engineering, where it is derived; nothing derives it here, no dataset supports the particular curve, and m is not an operationally defined quantity you could go and measure. It is in the book because the shape encodes something true and hard to state as forcefully in prose — that over-matching costs you as surely as under-matching — and because the symmetry is memorable at three in the morning. Treat the numbers it produces as illustrations of that asymmetry, never as predictions, and do not put them in a report.

At $m = 3$ — a method three times more sophisticated than the target requires — the factor is 0.75: you are throwing away a quarter of everything by being too clever, and the quarter comes out of you. At $m = 1/3$, the same. The novice's error is $m \ll 1$ (grepping a target that needed symbolic reasoning). The intermediate's error, which is more expensive because it is better disguised, is $m \gg 1$: building an elaborate structure-aware protobuf harness for a library whose bugs were three off-by-ones in an error path that a morning of reading would have found.

The practical content of (6.1) is that *effort and match are not substitutes*, which is the reason this Institute can make researchers out of ordinary talents: match, unlike raw ability, is learnable, so your aptitude for this work is the beginning of your story rather than its verdict. Raising E scales what you get and what you spend in a fixed ratio; only m moves the ratio. When a week has produced nothing, the instinct is to add a week. The mathematics is merciless and liberating at once: the exit from that spiral is never more effort, it is better match.

How do you estimate m before spending the effort? You reconnoitre. An hour spent reading the project's issue tracker, its last two years of security commits, its existing fuzzers, and its OSS-FUZZ status tells you most of what you need: if the project has been continuously fuzzed for four years, m for a naive byte-mutation campaign is approximately zero and you should be reading code, fuzzing the *encoder*, or doing variant analysis on its own past bugs. If the project has never been fuzzed, the opposite. Candidates skip the hour and spend the fortnight.

6.3 Redline: the cost of total coverage

Now the reckoning for the instinct Chapter 3 encouraged. Faced with a codebase, the conscientious response is to audit all of it, and the conscientious response is wrong. Total coverage of a nontrivial codebase is not achievable by any team you will ever be on; what attempting it produces is a uniform, shallow pass in which nothing is understood deeply enough to yield anything, plus a report whose findings section says "time did not permit".

The mature alternative is *scoped assurance*: choose a threat model, state it, and deny narrowly. Pick the two surfaces that are unauthenticated and parse attacker-controlled structure; go deep enough on those to understand the invariants; and write down explicitly, in the report, the surfaces you did not examine. A finding list of four real bugs plus an honest exclusions section is worth more than sixty pages of uniform shallowness, and the people who commission this work know it even when their statement of work does not say so.

Candidates find scoped assurance emotionally unsatisfying and then find it satisfying for approximately the rest of their careers. An engagement, like a specification, is strong in proportion to how little it claims.

6.4 Fatigue, and the sustainable-effort bound

The dissipation in the component is physiological, and unlike most of what a profession tells itself about stamina, this part has been measured. Let $P(t)$ be the intensity of sustained deep attention and F the accumulated fatigue:

$$\frac{dF}{dt} = kP(t) - r(F), \tag{6.2}$$

with $r(F)$ a recovery function, roughly linear at low fatigue and saturating as F grows, because recovery is a biological process with a ceiling. Researchers fail when F reaches a critical F_c, and the failure is not graceful. Write the bound this implies: sustained effort over a window of length Δ is limited by

$$\int_{t}^{t+\Delta} P\,dt' \le H(\Delta), \qquad H \text{ concave in } \Delta, \tag{6.3}$$

with concavity the clinical heart of it. *Sprint capacity does not scale to campaign capacity.*

Concavity is the load-bearing claim, so here is where it comes from. Pencavel's re-analysis of the British munitions data measured output against hours directly: below a threshold, output is proportional to hours worked; above it, output rises at a decreasing rate, and beyond roughly a forty-nine-hour week the additional hours buy so little that they are close to free to give up.[2] That is H concave, measured on real production by people who were not being asked their opinion about it. Van Dongen and colleagues supply the recovery half. Restricting sleep to four or six hours for fourteen consecutive nights produced cumulative, dose-dependent deficits across every cognitive task they ran. That is a saturating $r(F)$ rather than a linear one. And See and colleagues' meta-analysis establishes the within-session version: perceptual sensitivity itself declines with time on task, across a very large number of studies and task types.

None of that was collected on vulnerability researchers, and I am not going to pretend it was. What it establishes is that the shape is not a metaphor. A researcher who can hold a crisis analysis for nine hours cannot hold one-tenth that intensity for ninety days, and the reason is not character.

EDITORIAL NOTE

What is measured here and what is modelled. (6.2) is a model, written to make the argument tractable; nobody has fitted it to data on this profession, and k, r and F_c are not quantities you could go and measure about yourself this afternoon. (6.3) with H concave is the part with evidence behind it, and the evidence is cited in the notes rather than attributed to this Institute. Earlier printings called (6.3) "the Okoye limit" and credited it to eleven years of engagement logs across four laboratories. There were no logs. The name and the provenance were part of this book's framing and have been withdrawn; R. Okoye remains a character in it, and is no longer cited as a source. Theorem 6.1 was never affected either way, being ordinary mathematics about concave functions.

Theorem 6.1 (Rotation) *If H is strictly concave with $H(0) = 0$, then $2H(\Delta) > H(2\Delta)$: two researchers each working Δ at full sustainable intensity deliver strictly more than one working 2Δ.*

[2] Munitions work in 1915 is not code reading in 2026, and I would not lean on the exact threshold for anything. The shape is the part that transfers.

Proof. Strict concavity with $H(0) = 0$ gives $H(\Delta) = H(\tfrac{1}{2}\cdot 2\Delta + \tfrac{1}{2}\cdot 0) > \tfrac{1}{2}H(2\Delta) + \tfrac{1}{2}H(0) = \tfrac{1}{2}H(2\Delta)$. □ □

The practical reading of concavity is that sustainable *intensity*, $H(\Delta)/\Delta$, falls as the duration Δ grows. So engagements are planned inside the envelope with margin, and heroism is the practice of operating outside it exactly once. Every rotation policy, every mandatory-leave rule, and the regulation candidates resent most, the one forbidding analysis work the day after a disclosure crisis, is a corollary of that one fact about the shape of a curve.

6.5 What “talent” turns out to be

Both of the facts below belong in your self-model, and they belong there together, because each corrects the other’s temptation.

First: among people capable of the work at all, raw ability is a weak predictor of career output. What dominates is method — the reconnaissance habit of §6.2, the documentation habit of §3.5, the calibration habit of Chapter 8 — and method is learned. The most productive researcher I have worked with was, by every intake measure, unremarkable; he was also the only person in the building who read the coverage report every single morning, for eleven years, including the mornings when it had not changed.

Second: the tail is real. There exist a handful of people per generation whose pattern-recognition over code makes ordinary doctrine inapplicable, and pretending otherwise in front of candidates who have met one is dishonest. What I will say is that the tail is not where the field’s output comes from. The field’s output comes from thousands of methodical people filing thousands of well-written reports, and the mathematics in this book does not ask how gifted you are. It asks what you reached, and it bills accordingly, and in many years I have never met the researcher for whom equation (6.1) made an exception.

6.6 Substitutions: pipelines, analysers, and agents

Search need not be performed by continuous personal attention, and the profession’s history is a history of substitutions: fuzzers that run in CI on every commit; static analysers whose queries encode a departed colleague’s judgement; continuous services that fuzz a thousand projects at institutional scale; and now language-model agents that read code, propose hypotheses, and in 2024–2026 began reporting genuine memory-safety defects in SQLITE, OPENSSL, and the kernel — in at least one case a bug the vendor believed was already known to attackers.

The engineering rule for all substitutions is the same and is Vance’s:

Principle 6.2 (The substitution rule) *A substitution replaces* capacity*, never* judgement*. Automation searches the space it was pointed at, straight through every circumstance in which a human would have noticed that the space was wrong.*

The failure mode this predicts is the one every mature programme actually has: four years of green CI fuzzing, full dashboards, no findings, and a serious bug sitting in the encoder because the harness only ever exercised the decoder — a real and recurring pattern in long-enrolled continuously fuzzed projects. Nothing was broken. Nothing was watched, either.

Doctrine therefore staffs every substituted search with a *standing review*: one qualified person, not doing the searching, doing the noticing — reading the coverage delta, asking what the harness stopped reaching after last month’s refactor, asking what the agent’s report would look like if it were wrong. Budget

that review at roughly a tenth of the effort of the search it oversees, and place it inside the envelope of (6.3), because judgement fatigues on the same concave ledger as everything else about us. The reviewer who stops on schedule is maintaining the one component in the programme that cannot be reprovisioned.

NOTES AND SOURCES

This chapter carries more of the author's judgement than any other, and the reader is entitled to know where the line falls. (6.1) and (6.2) are models, flagged as such where they appear, and no data was fitted to either. Theorem 6.1 is ordinary mathematics about concave functions and needs no evidence beyond its proof. The chapter's one real empirical claim is therefore the concavity of H, which this printing cites rather than asserts.

Pencavel [36] is the direct evidence. Re-analysing the British Health of Munition Workers Committee data, he finds output non-linear in hours: proportional below a threshold, rising at a decreasing rate above it, with very little additional output beyond about a forty-nine-hour week. That is H concave, measured on production rather than on self-report. Van Dongen et al. [38] supply the recovery side: fourteen consecutive nights restricted to four or six hours produced cumulative, dose-dependent deficits across every neurobehavioral measure they took, which is the saturating $r(F)$ of (6.2) rather than a linear one. The same paper reports the finding that Exercise 6-3 turns on: subjective sleepiness ratings responded acutely and then rose only slightly, so subjects were largely unaware of deficits that kept accumulating. See et al. [37] give the within-session version, a meta-analysis establishing that perceptual sensitivity itself declines with time on task.

Two honest limitations. None of that work studied vulnerability researchers, or knowledge work of this particular shape, and the extrapolation from munitions output and psychomotor vigilance to code reading is mine rather than theirs. And concavity of H is what these sources support; the specific functional forms in (6.2), and the constants k, $r_{\max}$ and F_c, remain modelling conveniences with nothing behind them. Earlier printings attributed (6.3) to an "Okoye limit" assembled from eleven years of engagement logs across four laboratories. That provenance was invented and has been withdrawn.

The reconnaissance advice in §6.2 is independently supported: projects enrolled in continuous fuzzing for years still yield to a researcher who writes one harness for a path the existing harnesses never touched, and the encoding side of a codec is the classic case [19]. On substitutions, the agentic results referenced are [22] and [24].

The claim in §6.5 that method dominates raw ability, among people capable of the work at all, is the author's observation over a career rather than a measured result. The literature on variation in programmer productivity is contested enough that it should be read as such.

Exercises

Starred exercises require a licensed range, a written scope, and a supervising Second Bench. Unstarred exercises require a text editor. Do not confuse the two.

Exercise 6-1. From (6.1), compute the yield degradation at $m = 3$ and at $m = 1/3$, and state the result as a sentence a tired candidate could use at 3 a.m. Then describe, in two sentences each, a concrete engagement in which you would be at each value.

Exercise 6-2. You are given a library that has been in a continuous fuzzing service for four years with no

open findings. Estimate m for (a) a byte-mutation campaign on the existing harnesses, (b) a new harness for the encoder, (c) variant analysis over the project's own fixed bugs, (d) a manual audit of the error paths. Rank them, and justify the ranking by reference to §4.6 and §6.2 rather than to taste.

Exercise 6-3. (6.2) has nothing fitted to it. This exercise asks what it would take to believe it anyway.

(a) Take the model as given and work out what it says. Solve (6.2) for constant load P_0 with linear recovery $r(F) = F/\tau_r$; find the equilibrium fatigue and the condition for it to sit below F_c. Then let recovery saturate, $r(F) = r_{\max}\tanh(F/F_s)$, and show that no equilibrium exists above the critical load $P^* = r_{\max}/k$, that below it $F^* = F_s\,\mathrm{artanh}(kP_0/r_{\max})$, and that dF^*/dP_0 diverges as $P_0 \to P^*$.

(b) State what that divergence predicts about a person, in one sentence containing no symbols. Then split the prediction in two: which warning signs would a researcher approaching P^* expect to feel, and which would they expect *not* to? The second list is the substantive claim, and it is the one that could be wrong.

(c) Now stop treating the model as physics. Design the study that would test the claim in (b). Specify the population, the load variable you would manipulate, the outcome you would measure, the duration, and the number of subjects you think it would need. Then say why nobody has run it. Your answer to that last part should name at least one obstacle that is not funding.

(d) Read the Van Dongen result described in this chapter's notes and say, in two sentences, what it does and does not establish about your answer to (b). It is the closest measurement that exists, and it was not taken on this population, on this load variable, or over this duration.

Part (a) is not evidence for anything, and neither is the care with which you do it. The discipline of §8.5, which is separating what you have demonstrated from what you have assumed, applies to the apparatus in this book on exactly the terms it applies to a finding you are about to file.

Exercise 6-4. Using strict concavity of H with $H(0) = 0$, prove Theorem 6.1, then compute the gain factor $2H(\Delta)/H(2\Delta)$ for $H(\Delta) = H_0\Delta/(1+\Delta/\Delta_0)$ and evaluate at $\Delta = \Delta_0$. You should get $3/2$. You have just derived the rotation policy. File it under equations that are on your side.

Exercise 6-5. An automated pipeline has run for two years and reported nothing. Write the one-page standing-review checklist that would have caught the encoder-harness failure of §6.6. It must contain at least one item that is a question about *coverage* and at least one that is a question about *the diff*.

Exercise 6-6. * (Supervised; final-term candidates.) Keep a time log for four weeks recording, per session, hours spent and category of activity, and separately record every finding with the activity that produced it. At the end, compute your own realised yield per hour by category. You will find the distribution is not the one you expected, and that it disagrees with where you spend your time. The plot, initialled, goes into your training file, where someone with my job will show it to you again on the day you argue with a rotation order.

CHAPTER 7

Five Standard Engagements

Doctrine is what survived.

— Inscribed above the issue desk, Ninth Street stores. Nobody claims it; everybody quotes it.

BY THE END OF THIS CHAPTER

Recognise which of the five standing engagement shapes a situation calls for, plan it, and say in advance how it will fail. Adjust the plan for binary-only targets, where every instrument gets harder by one level of indirection. And for work on live systems, treat authorisation, data handling and blast radius as things established on paper before the first execution rather than managed by intention.

7.1 How to read a pattern

This chapter descends from method to the five standing engagement shapes every Systems candidate must be able to plan, execute, and hand over. For each I give the purpose, the method by citation to Chapters 2–6, the characteristic failure modes, the conditions under which it is the wrong pattern, what finishing looks like, and a doctrine table. A pattern is not mastered when you can run it. It is mastered when you can say, before it fails, how it will fail, and when you can say why you did not choose one of the other four. The five are presented in ascending order of what a mistake costs.

Choosing between them is mostly a matter of answering three questions in order, and the order matters more than the answers. *Do you have source?* No sends you to Pattern IV, and the honest follow-up is whether you could have obtained source by asking, because a surprising amount of binary work is done out of habit rather than necessity. *Is the system live, and not yours?* Yes sends you to Pattern V, and Pattern V is almost always the wrong answer; the question underneath it is whether a lab replica would answer what you actually want to know. Otherwise you are choosing on duration and on who owns the result afterwards: days with nobody owning it afterwards is Pattern I, indefinite with a named owner is Pattern II, and months with a client and a report of record is Pattern III. Reconnaissance (§6.2) comes before all of this, because the estimate of m it produces is what tells you whether the pattern you have chosen has any yield in it at all.

A word about the tables. The numbers in them are my working defaults, not measurements, and earlier printings called them “conservative composites over many engagements”, which credited them to a body of data that does not exist. They are chosen so that a tired candidate meeting an unfamiliar codebase on a Monday can plan inside the envelope without computing anything. Where a row restates documented practice, the practice is cited in the notes and you can go and check it. Where a row gives a reporting reserve or a review cadence, it is a starting point, and the first thing a competent programme does is replace it with something it measured itself. You now know enough to beat these tables on a good day with

a well-scoped target. Doctrine is not for good days.[1]

7.2 Pattern I: the drive-by

Purpose. A short, bounded look at a small library — a weekend, a hackathon, the gap between two client engagements. Not an assurance activity and honest only about what it touched.

Method. Reconnaissance first (§6.2): read the tracker, the last two years of security commits, and whether the project is already continuously fuzzed. Then the cheapest high-m instrument, which for an unfuzzed small library is almost always a single **LLVMFuzzerTestOneInput** over the obvious entry point with a sanitiser build and the project's own test files as seeds. Two hours to first execution or you have chosen the wrong target. Manual reading (§3.5) fills the waiting.

Failure modes. The target was already fuzzed and you spent the weekend confirming it ($m \approx 0$; the reconnaissance hour would have told you). Harness over-reach (§5.3) producing a pile of artefacts. And the characteristic one: finding something real on Sunday night and reporting it badly on Sunday night. A drive-by has no reporting time budgeted, so budget it: if you have four hours left, spend three on Chapter 8 and one on looking further.

When it is the wrong pattern. When reconnaissance says the project has been continuously fuzzed for years, because the cheap instrument that defines this pattern is the one with $m \approx 0$ against such a target, and what you want is Pattern III's reading or a harness for a path nobody wrote one for. When you cannot finish reporting inside the window. And when the target is not yours and you hold no permission, at which point there is no pattern, there is only §1.4's range safety box.

What done looks like. Either a reported finding with a reproducer that meets Chapter 8's bar, or a written paragraph saying what you reached, what you did not, and what you would do with another two days. Never neither. The second outcome is the common one and it is not a failure; a drive-by that ends with the harness and corpus committed somewhere a stranger can pick them up has produced something, and one that ends with a browser tab closed on Sunday night has not.

Parameter	Doctrine value	Section
Reconnaissance	1 hr, before any tooling	§6.2
Time to first execution	$\leq$ 2 hr	§5.3
Build	ASan+UBSan, always	§2.5
Seeds	project test suite, minimised	§4.5
Reporting reserve	40% of remaining time	Ch. 8

7.3 Pattern II: the harness campaign

Purpose. The default answer, and the one that produces most of the field's output: build harnesses, integrate them into continuous infrastructure, and let them run indefinitely.

Method. One harness per logical entry point, each deterministic, bounded, and leak-clean (§5.3). Corpus committed to the repository and minimised weekly (§4.5). Build matrix across plain, ASan+UBSan,

[1] Nor is it for arguing with. If you find yourself explaining to a reviewer why the reserve did not apply to your engagement, you have discovered why the reserve is a number and not a judgement call.

CmpLog, and MSan where affordable, with corpora synced (§4.5). Dictionary per format. Coverage report generated and *read* on a schedule (§5.1) — this is the standing review of §6.6 and the line item that gets cut first and should not be. New harnesses added when coverage says a subsystem is unreached, not when somebody has an idea.

Failure modes. Harness rot: a refactor changes an entry point, coverage silently drops 30%, and nobody notices for a year. Decode-only bias: nobody fuzzed the encoder, the serialiser, or the error paths. That is where the surviving bugs in mature projects actually live. And plateau-as-conclusion, the belief that a flat coverage curve means an absence of bugs rather than an absence of reach (§5.1).

When it is the wrong pattern. When nobody will own the coverage review. A campaign whose report is generated and never read is not Pattern II, it is a dashboard, and Principle 6.2 predicts precisely what it will miss. If you cannot name the person and the cadence, you are proposing something else and should say so. Also when the bug class you are actually chasing does not crash: differential and round-trip oracles (§5.4) are still Pattern II, but a campaign whose only oracle is a sanitiser will run for years past a parser disagreement without noticing it.

What done looks like. This pattern does not finish. That is why an exit criterion matters more here than anywhere else. Substitute a standing one: a harness is in good order when the named reviewer can state, from the last coverage report, which subsystem it does not reach and why that is acceptable. A campaign that cannot produce that sentence has stopped being evidence and has become infrastructure.

Parameter	Doctrine value	Section
Harness determinism	absolute; no time, net, threads	§5.3
Corpus minimisation	weekly	§4.5
Build matrix	$\geq$3 configurations, synced	§4.5
Coverage review	monthly, by a named person	§6.6
Encoder coverage	explicitly tracked	§4.6
Regression corpus	every fixed bug, forever	Ch. 8

7.4 Pattern III: the deep audit

Purpose. Sustained assurance on one codebase over months: a security review of record, the thing a serious organisation does before shipping something it cannot recall. The full apparatus of this book, applied with patience.

Method. Attack-surface enumeration and trust-boundary documentation first, written down and agreed with the client (§3.5) — the document is the deliverable's spine and the defence against scope drift. Backward walks from every sink class, producing a written provenance table. Variant analysis over the project's own fixed bugs (§3.6). Harnesses for what is fuzzable and reading for what is not. Explicit exclusions, stated and justified (§6.3). A mature deep audit produces three artefacts: the findings, the exclusions, and the invariant list — and the invariant list, handed to the maintainers as assertions they should add, is frequently worth more to them than the findings.

Failure modes. Uniform shallowness (§6.3), the commonest and the most expensive. Scope drift, defeated only by the boundary document. Auditor fatigue against the concavity of H (§6.4), which on a four-month engagement is not a metaphor. And the institutional one: the review that becomes an annual

ritual whose findings section shrinks every year because nobody is willing to say that the same two surfaces have been re-examined four times.

When it is the wrong pattern. When the scope cannot be narrowed. A client who will not accept an exclusions section is asking for uniform shallowness with a covering letter, and §6.3 is the argument to have with them before signing rather than after delivering. Decline, or reduce the claim to a Pattern II engagement plus a written threat model, and say which you did.

What done looks like. All three artefacts exist and the exclusions section is longer than the client expected. That last part is the test. An audit whose exclusions fit in a paragraph either examined a very small thing or is not telling you where it did not look, and a reader who cannot tell which is holding a document that establishes nothing. Add one more check before delivery: every finding states ρ, χ and κ separately, and says which of the three is weakest.

Parameter	Doctrine value	Section
Boundary document	before analysis; counter-signed	§3.5
Surfaces examined	few, deep; exclusions written	§6.3
Provenance table	every sink, every operand	§3.5
Variant sweep	on all prior fixed bugs	§3.6
Rotation	inside the sustainable-effort envelope	§6.4
Deliverables	findings + exclusions + invariants	Ch. 8

7.5 Pattern IV: the moving target

Purpose. Binary-only work: firmware, a closed-source library, an appliance, a driver blob. The hardest routine pattern, and the one where the methods of Chapters 3 and 4 are least directly available, because you have neither source to read nor a compiler to instrument.

Method. Everything gets harder by one level of indirection. Coverage comes from dynamic binary instrumentation or emulation rather than from the compiler — AFL++'s QEMU, FRIDA, and Unicorn modes exist precisely for this, at a documented throughput cost of two to five times, considerably better than the general-purpose instrumentation frameworks and recoverable in part by persistent mode. The compiler's sanitisers are unavailable, so unless you stand up binary-only shadow-memory tooling the oracle degrades from "corruption" to "signal". Silent memory errors go unseen, and your findings skew towards crashes that happen to fault. Compensate with binary-only shadow-memory tooling where the architecture allows it, and with Valgrind where you can run the code at all. Reversing replaces reading: recover the parser's structure well enough to write a dictionary and a structure-aware generator, since these matter more here than anywhere else. You cannot afford to waste executions when each one costs several times what it costs on a source build.

Failure modes. Throughput collapse, and the associated temptation to draw conclusions from campaigns three orders of magnitude smaller than Chapter 4's arithmetic requires. Oracle blindness: no sanitiser means the corruption you found is the corruption that faulted, and the ratio is unfavourable. Emulation

infidelity — the bug is in your emulator, not the target, a failure that has consumed entire engagements. And the architectural trap: reproducing on your workstation what only occurs on the device.

When it is the wrong pattern. When source exists and you did not ask for it. Vendors under a disclosure programme, and a great many firmware suppliers under a contract, will hand over source to a named researcher for the asking, and the hour spent asking is the highest-*m* hour available in this entire pattern. Ask first. Also when the target is available to you only as a running service rather than as an image: Pattern V, and a different conversation entirely.

What done looks like. The finding reproduces *on the device*, not only under emulation, and you have run at least one differential check of emulator against device on inputs that do not crash, so that the reproduction is evidence rather than coincidence. State the throughput you achieved and the resulting bound on p from (4.3). Against a binary-only target that bound is weak, and a report that hides how weak is worse than one that prints it.

Parameter	Doctrine value	Section
Coverage source	DBI or emulation; measured	§5.1
Throughput expectation	0.2–0.5× source-build	§4.3
Oracle	weakest available; state it	§2.5
Input model	structure-aware, mandatory	§4.4
Reproduction	on-device before reporting	§8.4
Emulator sanity	differential vs device	§5.4

7.6 Pattern V: the live target

Purpose. Kernels, hypervisors, privileged services, and anything where a mistake during testing costs somebody else something. The pattern where every margin in this book doubles, because the party bearing the risk of your error is not you.

Method. Nothing runs on a machine you would miss. Kernel work happens inside disposable virtual machines under a structure-aware system-call generator with declarative interface descriptions, coverage from the kernel's own instrumentation, and the kernel sanitisers as the oracle (§5.2) — this is the **syzkaller** arrangement and there is no credible substitute for it. Snapshot and restore rather than reboot. Resource types matter: a generator that does not know which call produces a file descriptor and which consumes one will spend its life failing at **EBADF**.

And the part that is not technique. Authorisation is a document, it names the target, the dates, the techniques, and the people, and it exists before the first execution. Data handling is decided in advance: what you may see, what you must not retain, what you do with credentials you stumble across. Blast radius is bounded structurally — separate hardware, separate network, separate credentials — and not by your intention to be careful.

Failure modes. Scope creep during an engagement, which is how a lawful test becomes an unlawful one without anybody deciding to break the law. Shared infrastructure: the campaign that filled a shared filesystem or saturated a shared link (§4.6). Incidental data exposure, for which the only defence is having written down beforehand what you would do. And the long-hold problem: a temporary authorisation that everyone treats as permanent because nobody wants to re-open the conversation.

When it is the wrong pattern. Almost always, and the question that settles it is whether a lab replica would answer what you want to know. Usually it would, at a fraction of the risk, and the reason people reach for the live system anyway is that building the replica is a fortnight of unglamorous work. Pattern V is correct when the property under test is a property of the deployment rather than of the software: a configuration, an integration, a piece of state that exists only in production. If you cannot write down which of those it is, you want Pattern III against the same code.

What done looks like. The authorisation was in date on the day of every execution, the blast radius held, and the data-handling procedure was followed rather than merely written. Those are three factual claims about the engagement, and the deliverable states them alongside the findings. A live-target engagement that produced findings and cannot say those three things has not finished; it has stopped.

Parameter	Doctrine value	Section
Authorisation	written, specific, dated, signed	Ch. 1
Execution environment	disposable VM; snapshot restore	§5.2
Interface model	declarative, with resource types	§5.2
Oracle	kernel sanitisers, always on	§2.5
Blast radius	bounded by hardware, not care	§4.6
Incidental data	procedure written in advance	§8.7
Authorisation renewal	explicit; never assumed	Ch. 1

RANGE SAFETY

Candidates may run Patterns I and II on their own code or on public projects under the Institute's standing open-source policy, may participate in Pattern III under a qualified lead, and may *observe* Patterns IV and V. The restriction on Pattern V is absolute and is not about your competence. An unauthorised test of a live system is, in most jurisdictions, an offence that turns on access rather than on harm; a well-intentioned test that takes down a hospital's imaging server is still the thing that took down the imaging server; and the person who signs the authorisation is accepting professional consequences on your behalf, which is a courtesy you repay by never once exceeding what the paper says.

NOTES AND SOURCES

Pattern II is essentially the model OSS-Fuzz documents and operates [35], and where a row of its table restates that practice — corpus in the repository, minimisation on a schedule, a build matrix with synchro-

nised corpora, a dictionary per format, coverage reports generated and read — the practice is theirs and not this Institute's. The decode-only bias named as Pattern II's characteristic failure is documented in [19].

The throughput expectation for Pattern IV is from the AFL++ documentation [3]: two to five times slower than compile-time instrumentation for the QEMU and FRIDA modes, recoverable in part by persistent mode. Earlier printings said an order of magnitude, which overstated the penalty by enough to change the arithmetic in Exercise 7-4.

Pattern V's technical arrangement is syzkaller's [50]. Its legal content is not technique, and is discussed with the rest of the legal material in the notes to Chapter 8.

Exercises

Starred exercises require a licensed range, a written scope, and a supervising Second Bench. Unstarred exercises require a text editor. Do not confuse the two.

Exercise 7-1. You have five days and a 60,000-line C library that has been in a continuous fuzzing service for three years with no open findings. Choose a pattern, justify the choice by reference to §6.2, and write the five-day plan including what you will not do.

Exercise 7-2. For Pattern II, write the monthly standing-review checklist. It must contain one item about coverage delta, one about the diff since last review, one about harness determinism, and one about the encoder. Then explain, in two sentences, why a checklist without a named owner is not a control.

Exercise 7-3. A client asks for a Pattern III deep audit but will not agree a boundary document because "we want you to look at everything". Draft the two-paragraph reply. It must be one a client would accept.

Exercise 7-4. For Pattern IV, compute what a $10\times$ throughput penalty does to the campaign budget of Worked Example 4-1, and state the smallest p you could still expect to find in three days. Then name the two techniques that would recover the lost ground, and why neither is optional.

Exercise 7-5. Draft the incidental-data procedure for a Pattern V engagement: exactly what you do, in order, in the first ten minutes after a fuzzing artefact contains what appears to be a real user's personal data. One page. The exercise is the ordering.

Exercise 7-6. * (Supervised.) Under a Second Bench, stand up a Pattern V rig on your own hardware: a kernel built with coverage and sanitiser instrumentation, a disposable VM, and a structure-aware system-call generator reaching one driver you have chosen. You are not hunting. You are demonstrating that you can build an environment in which a crash costs nothing, because for the rest of your career the quality of your rig will set the ceiling on what you are permitted to touch.

CHAPTER 8

The Proof of Concept

> *A crash is a rumour. A reproducer is a fact. An impact claim is an argument, and arguments can be lost.*
>
> — R. Okoye, retirement lecture, 2021.

BY THE END OF THIS CHAPTER

Reduce a crashing input until every byte is load-bearing, and know what 1-minimality does and does not promise. Work backwards from the crash site to the defect site, and state the root cause as a single sentence naming the assumption that failed. Package evidence so a stranger reproduces it on the first attempt. Describe a primitive instead of asserting an outcome. And choose a disclosure policy before you need one.

8.1 What a proof of concept is for

A proof of concept is evidence. Its purpose is to transfer a fact from your machine to somebody else's with no loss and no dispute: that this input, against this build, produces this misbehaviour, at this line, for this reason. Every property we will demand of it (minimality, determinism, self-containment, an honest impact section) follows from that single purpose and from nothing else.

Let me be explicit about what a proof of concept is *not*, because the word carries freight from an adjacent discipline. It is not a working exploit. Weaponising a memory-safety bug against a modern target is a genuine speciality with genuine practitioners: defeating a canary, leaking an address to defeat layout randomisation, grooming an allocator into a favourable state, chaining gadgets past control-flow enforcement. Essentially none of it is necessary to get a bug fixed. A maintainer needs to reproduce the defect and understand its class. A patch-prioritisation decision needs an honest severity argument. Neither needs a payload, and the discipline this Institute teaches is that we produce the reproducer and the argument and we stop there.

Rule 8.1 (The evidence standard) *A proof of concept is complete when a competent stranger, given only your artefact, can reproduce the misbehaviour on the first attempt and can locate the defective line without asking you a question.*

8.2 Minimisation

A fuzzer's crashing input is typically large, structurally corrupt in a dozen irrelevant ways, and useless as evidence. The first act of triage is to shrink it until every remaining byte is load-bearing.

Mechanically, use the tools: **afl-tmin** removes blocks while preserving the observed behaviour and gets you most of the way in seconds; libFuzzer's **-minimize_crash=1** does the equivalent. *Where the tools stall*, meaning checksums, compressed sections, and formats where deleting a byte invalidates a length, fall back to delta debugging by hand or to a format-aware reducer that understands the structure.

The underlying algorithm is worth knowing rather than invoking, because you will need to implement it for structured inputs. Delta debugging (*ddmin*) partitions the input into *n* chunks, tests each chunk's complement, and recurses: if removing some subset still crashes, continue with the smaller input; if no subset works at granularity *n*, double *n* and retry, down to single elements. The result is *1-minimal*: no single remaining element can be removed. Note what that does not say: 1-minimal is not globally minimal, and two different reductions of the same crash can differ.

Minimisation is not cosmetic. It usually collapses several apparent crashes into one, which is the only honest de-duplication (§4.6). It frequently reveals the bug, since an input reduced from 40 KB to 14 bytes often has the defect legible in it. And it makes the artefact citable. A maintainer will read fourteen bytes and will not read forty kilobytes.

WORKED EXAMPLE 8-1 From 92 kilobytes to nine

A campaign against a font parser reports 3,100 crashing inputs. Do not triage 3,100 things. Run **afl-tmin** over all of them in parallel; the median input drops below 200 bytes and several hundred collapse to byte-identical files. Group what remains by ASan error class plus the top three frames of the stack, remembering that this bucketing over-counts and occasionally under-counts (§4.6) and is therefore a work-ordering device and not a bug count. Take one representative per bucket — now perhaps eleven — and minimise those by hand, format-aware. Two reduce to the same nine-byte table header with an implausible entry count, differing only in a field the parser ignores: one bug, not two, and its nine bytes say what it is. Elapsed: an afternoon. Reporting 3,100 crashes would have taken a month and been wrong. ▲

8.3 Root cause: the crash site is not the bug site

Now the central skill of this chapter, and the one that separates a report a maintainer acts on from one they close. The place where the program died is very often not the place where it went wrong. An allocator abort is the allocator discovering damage inflicted earlier. A use-after-free faults at the *use*, and the bug may be in the **free**, or in the refcount decrement that preceded it, or in the error path that took the reference away. An overflow faults where the write lands, and the defect is usually where the *size* was computed: the sixth shape from §3.3.

The technique is to work backwards along the object's life, and the tools are:

The allocation and free stacks. ASan hands you these free (§2.5). Read them before you open a debugger. For an overflow, the allocation stack tells you where the size was decided, and comparing the decided size with the written size is very often the entire investigation.

Watchpoints. Break at allocation, take the address, watch it, run. The write that sets the bad value is the bug, and it is frequently in a function that does not appear in the crash stack at all.

Reverse execution. Record-and-replay debugging (**rr** on Linux, where the architecture supports it) lets you set a watchpoint on the corrupted memory and run *backwards* to the instruction that wrote it. For a non-trivial corruption bug this turns a two-day investigation into a twenty-minute one, and candidates who learn it early outperform candidates who do not for years.[1]

[1]The setup cost is real and everybody quotes it as a reason not to bother. It is about an hour, once.

Bisection. If the defect is a regression, **git bisect** with your minimised input as the test gives you the introducing commit, and the introducing commit usually contains the author's reasoning in its message. This is the cheapest root-cause technique that exists and the most neglected.

The disassembly. Because of §2.3, the check you read in the source may not be in the binary. When source-level reasoning says a bug is impossible and the crash says otherwise, the compiler is the third party in the conversation.

Stop when you can state the defect as a sentence of the form: *at **file:line**, X is computed under the assumption A, which does not hold when C, because B*. If you cannot write that sentence, you have a crash and not a finding, and the honest thing is to say so in the report.

8.4 The artefact

Package the evidence so that reproduction is mechanical. The standard bundle:

- **The input**, minimised, as a file, with its length and a hash. Not pasted as hex into an email body, where a mail client will mangle it.
- **The build**, exactly: compiler, version, flags, sanitisers, target commit hash. "Latest master" is not a build.
- **The command**, verbatim, one line, no shell aliases.
- **The observed output**, verbatim: the full sanitiser report, unedited, including the frames you think are irrelevant.
- **A container recipe** where the build is non-trivial. A ten-line **Dockerfile** that reproduces from a clean base is the single highest-value item in the bundle and the thing most likely to get your report actioned this week rather than next quarter.
- **A regression test**, in the project's own format, ready to merge. Maintainers notice this. It is also the thing that stops the bug coming back.

```
# poc/run.sh -- reproduce GHSL-style finding on a clean machine
git clone https://example.invalid/target && cd target
git checkout 4f2a9c1
clang -g -O1 -fsanitize=address,undefined -o fuzz_parse \
      fuzz/fuzz_parse.c src/*.c -Iinclude
./fuzz_parse ../poc/min.bin      # expect: heap-buffer-overflow WRITE of
    size 1
```

8.5 The impact argument, and its honest limits

Here is where calibration is earned or lost. Your severity claim should be an argument with premises, each of which you have either checked or explicitly not checked. The three premises are the three questions of §1.2.

REACHABILITY: state the interface, the privilege required, and the preconditions. "Reachable by any local user in the default configuration" and "reachable by a caller who has already authenticated and enabled a non-default option" are different findings.

CONTROL: state, in bits or bytes you can name, what the attacker chooses. *Offset:* fixed or controlled, and over what range. *Value:* fixed, partially controlled, or arbitrary. *Length:* fixed or controlled. *Repeata-*

bility: once, or many times without crashing. Repeatability is often the single most important variable, because a primitive you can invoke repeatedly is qualitatively stronger than one you cannot.

CONSEQUENCE: and here the discipline is to describe the primitive rather than to assert an outcome. "A one-byte out-of-bounds write with attacker-chosen value at a controlled offset into an adjacent heap allocation" is a claim you can defend. "Remote code execution" is a claim you have not established unless you established it, and asserting it without evidence is the fastest way to lose a maintainer's trust permanently.

Some specific honesties, none of which cost you anything:

1. **Say when you do not know.** "We did not assess exploitability; the primitive is described above" is a professional sentence. It is also what the automated exploitability classifiers are doing, less honestly. The well-known GDB **exploitable** plugin and its relatives are heuristics that are wrong in both directions often enough that experienced practitioners treat their verdicts as, at best, a work-ordering hint.
2. **Name the mitigations.** The severity of a heap overflow depends on the allocator, on whether the target is built with the relevant hardening, and on the platform. A bug that is a crash on a target with a hardened allocator and pointer authentication may be more on a target without them. Say which you tested.
3. **Distinguish your build from theirs.** If you disabled a checksum to fuzz (§4.6), the finding is not established against the shipped binary until you produce an input with a valid checksum. Either do that work or write the caveat. Failing to do either is the most common reason a real bug is closed as invalid, and the maintainer is not being difficult.

ON THE WORD "EXPLOITABLE"

Candidates want to write it, because it is the word that makes a finding feel important, and because somewhere there is a scoring rubric that rewards it. Consider what you are claiming. You are asserting the existence of a reliable path from this corrupted state, on a real target, past every deployed mitigation, to a specific attacker objective. That is a substantial engineering claim, it is usually weeks of work, and the honest answer for most findings is that nobody has done that work. Meanwhile the *useful* content of your report, the reproducer and the root cause and the primitive description, is complete without it. So write what you know: the primitive, its parameters, its repeatability, the mitigations in play. A reviewer who wants a severity number can compute one; a reviewer who catches you inflating one will discount everything else you ever send them, and the field is small enough that this matters for the length of a career.

8.6 The report

Structure, in the order a busy maintainer reads:

1. **One-line summary.** Class, component, and reachability. *"Heap out-of-bounds write in **parse_table()** reachable from any untrusted input file."*
2. **Affected versions and commit range**, with the introducing commit if known.
3. **Reproduction**, verbatim: build, command, input, observed output.
4. **Root cause**, as the sentence from §8.3, with the relevant source quoted minimally and a pointer to the line.

5. **Impact**, as the three-premise argument of §8.5, with the uncertainties stated.
6. **Suggested fix**, if you have one, offered as a suggestion. You know the bug; they know the codebase.
7. **Disclosure intentions**, dates included, stated plainly and without menace.
8. **Credit preference**, and how you wish to be named.

Tone: neutral, specific, short. You are handing a colleague a problem, not winning an argument. Most maintainers of the projects you will report to are unpaid, are receiving your report on a weekend, and are the reason the software exists at all.

8.7 Disclosure

Coordinated disclosure is the norm and the details are conventions rather than laws, so know the landscape and pick a policy *before* you find something.

The dominant pattern is a deadline with a grace period. Google's Project Zero publishes on a 90+30 schedule: ninety days for a patch, then details thirty days after the patch ships, with a fourteen-day grace extension if a vendor commits to a date, and a much shorter track when a bug is already being exploited in the wild: seven days, with a three-day grace period, and the same thirty-day post-patch window. CERT/CC's default is forty-five days from report, patch or no patch. CISA's coordination programme may disclose from forty-five days where a vendor is unresponsive. The reasoning behind deadlines is worth understanding rather than merely obeying: absent one, a non-trivial fraction of reports are never fixed, and the published outcome data supports the claim that deadlines move vendors. Project Zero reports that with the grace period available, the overwhelming majority of its issues were fixed inside the deadline.

Mechanics you should get right. Find the project's security policy first, meaning a **SECURITY.md**, a security address, or a bounty programme, and use the channel it names; reporting a memory-safety bug in a public issue tracker is a decision,[2] and usually the wrong one. Request a CVE identifier through the appropriate numbering authority, preferring the vendor's own where one exists, so the first public mention has a stable identifier attached. Keep a timeline: every contact, date, and response. Agree the embargo explicitly rather than assuming it. And if you intend to publish a write-up, agree the date. You should intend to. The field learns from write-ups, and yours will be better than my summary of somebody else's.

The harder cases. *Unresponsive vendors*: escalate to a coordinator rather than improvising, because a coordinator has leverage and standing that you do not. *Bounty programmes*: read the scope, and understand that a bounty's terms are a contract that may restrict your publication rights; decide whether you accept that before you submit, not after.

RANGE SAFETY

Three things a candidate has done and must not. *One*: attaching a working exploit to a first report "to prove severity". You have now created and transmitted a weapon, possibly across a border, for a benefit the reproducer already provided. *Two*: hinting at consequences to accelerate a response — mentioning a conference deadline, a journalist, or a price. Depending on wording and jurisdiction that is extortion, and the line is not where

[2] Some projects will tell you to file publicly anyway. Believe them. It is their tracker and their threat model, and arguing about it in the open is itself a disclosure.

an enthusiastic twenty-three-year-old thinks it is. *Three*: testing whether the bug affects somebody else's production deployment. You may test your own copy. Confirming reachability on a third party's live system is unauthorised access, and the fact that you were confirming a genuine vulnerability is not a defence anywhere I am aware of.

NOTES AND SOURCES

The reduction algorithm is *ddmin* [55], and the definition of 1-minimality used here is that paper's. §8.2 describes the reduce-to-complement rule; the published algorithm carries a second rule that recurses into a failing subset, a divide-and-conquer optimisation whose benefit depends on the shape of the failure and is not always positive. Either form produces a 1-minimal result. Record-and-replay debugging is [33].

The crash-classifier heuristics deprecated in §8.5 are the familiar debugger plugins; their unreliability in both directions is why §9.4 lists exploitability as an open problem rather than a solved one.

On disclosure, the 90+30 schedule, the fourteen-day grace period and the seven-plus-three in-the-wild track are Project Zero's published policy [23]; CERT/CC's default is [12], CISA's coordination practice is [13], and the process standards worth knowing are [26].

On the legal material. The Range Safety boxes state that unauthorised testing turns on authorisation rather than on intent or harm in most jurisdictions. That is a summary of a varied legal landscape, not advice, and it is no substitute for counsel in yours. Where those boxes mention export control in connection with transmitting exploit code, the regimes in question are real, but their scope is narrower and more contested than the sentence implies; read the rules that actually apply to you before relying on the paraphrase.

Exercises

Starred exercises require a licensed range, a written scope, and a supervising Second Bench. Unstarred exercises require a text editor. Do not confuse the two.

Exercise 8-1. Implement *ddmin* for byte inputs in the language of your choice, in under sixty lines, and verify it produces a 1-minimal input on a crash of your own making. Then construct an input on which your implementation returns a 1-minimal result that is not globally minimal, and explain why that is acceptable.

Exercise 8-2. You have an ASan **heap-buffer-overflow WRITE of size 1**, 8 bytes past a 24-byte region. List the facts you would establish, in order, before writing an impact section, and for each the tool or observation that establishes it.

Exercise 8-3. Rewrite each claim as something defensible: (a) "this is trivially exploitable for RCE"; (b) "we found 400 vulnerabilities"; (c) "the attacker controls the heap"; (d) "this is only a crash, so it is low severity".

Exercise 8-4. Write the complete report for a bug you have found in your own code, following §8.6. Then hand it to a classmate who has not seen the bug and time how long they take to reproduce it. If they ask you a single question, the report failed; fix it and repeat.

Exercise 8-5. Draft your personal disclosure policy: your deadline, your grace period, your in-the-wild track, your position on publication, and your position on bounty terms that restrict it. One page, written

before you need it. Then state which of your rules you expect to find hardest to keep, and why.

Exercise 8-6. * (Supervised.) Take a fixed, public memory-safety CVE. Independently reconstruct the minimised reproducer from the patch alone, root-cause it to the sentence form of §8.3, and write the impact argument without reading the original finder's advisory. Then read the advisory and write a one-page comparison: what you got right, what you missed, and where you were over- or under-confident. This exercise, repeated a dozen times, is calibration training, and calibration is the only thing in this profession that compounds.

CHAPTER 9

What We Do Not Know

An honest syllabus ends with its ignorance, arranged neatly, where the students can reach it.

— R. Okoye, retirement lecture, 2021.

BY THE END OF THIS CHAPTER

Name the five things this field cannot currently do, and say what each would be worth. Recognise which of your own conclusions rest on those gaps, particularly any claim about what a clean campaign establishes. Hold the question of automated discovery open, because the evidence is arriving faster than anyone's position on it.

9.1 Why this chapter exists

Every claim in Chapters 1 through 8 rests on measurement and survives use. This chapter is the other ledger: the things vulnerability research cannot yet compute, predict, or in one case respectably discuss, arranged neatly where you can reach them, because a defensive discipline that hides its open problems is indistinguishable, on the day it matters, from folklore with better tooling. Each problem is stated as sharply as I can state it and annotated with what a solution would be worth. The standing offer of the first edition remains: to any candidate who retires one of these, dinner on me and the renumbering of this chapter in the next edition.[1]

9.2 Problem 1: the harness gap

We cannot measure fidelity. Definition 5.2 defines ϕ as the correspondence between states a harness can construct and states an attacker can reach, and then this book uses the concept for three chapters without ever computing it, because nobody can. We have coverage, which tells us what we executed; we have no metric at all for what fraction of the *real* reachable state space our harnesses represent, and therefore no way to convert "four years of clean fuzzing" into a statement about residual exposure.

The practical cost is visible in the record: projects enrolled in continuous fuzzing for years still yield serious bugs to a human who writes one harness for a path the existing harnesses never touched, classically the encoding side of a codec whose decoder was fuzzed to death. Nobody was negligent. The instrument simply does not report the quantity that would have warned them. *Worth*: a defensible fidelity metric would turn fuzzing from an activity into an assurance argument, which is the difference between telling a regulator what you did and telling them what is left.

[1] The offer has been collected once, in 2019, for a problem that consequently no longer appears. The solution was three pages. The dinner was worth considerably less than three pages and I said so at the time.

9.3 Problem 2: residual defect estimation

Given a campaign's coverage curve, its duration, and its find history, we cannot estimate how many bugs remain. Coverage growth and bug discovery are strongly correlated in aggregate, and the correlation is useless at the level of a decision, because the relationship is mediated by the unmeasured ϕ of Problem 9.2 and by a defect distribution we cannot observe. The honest state of the art is that campaigns exhibit long quiet stretches punctuated by bursts when some coverage barrier falls, so a flat month predicts almost nothing about the next one.

Software reliability growth models exist and are applied to fuzzing with results that range from suggestive to embarrassing, largely because their independence assumptions are false in a search that deliberately correlates its own samples. *Worth*: more than any other open problem in the field, and less tractable than any of them. The ability to say "this component has an estimated n remaining reachable memory-safety defects, with these error bars" would restructure how security work is funded, scheduled, and argued about.

9.4 Problem 3: exploitability

There is no sound automated oracle for exploitability, and there is reason to think there cannot be a complete one: deciding whether a corrupted state can be driven to an arbitrary objective is, in the general case, a question about the reachable states of the whole machine. What we have are heuristics: the classifier plugins that bucket crashes as exploitable or probably-not. They are wrong in both directions frequently enough that experienced practitioners use them for work ordering and nothing else.

The consequence runs right through this book. It is why §8.5 asks you to describe primitives rather than assert outcomes; it is why severity scoring is contested; and it is why the industry's patch-prioritisation decisions rest, ultimately, on the judgement of individual researchers rather than on a computation. *Worth*: even a sound *negative* oracle, a procedure that could certify a crash as definitely not exploitable under stated mitigations, would remove an enormous amount of wasted defensive effort. I would send a doctoral student at the negative half, and I do not know whether they would come back with anything.

9.5 Problem 4: crashes to bugs

We cannot reliably map crashing inputs to distinct defects. Coverage-based uniqueness inflates one bug into hundreds; stack hashing does better and still over-counts substantially, while also silently *under*-counting, since a measurable fraction of hashes are shared between genuinely different bugs. The deduplication step in every pipeline in the world is therefore quietly discarding real findings. Root-cause clustering approaches do better and are not standard.[2]

This is the most embarrassing item on the list, because it is not a deep question about computation; it is a basic measurement problem in the middle of our most-used workflow, and it contaminates everything downstream. Every published comparison of fuzzers that counts "unique crashes" is measuring an artefact. Every campaign report with a crash count in it is overstating or understating by an unknown factor. *Worth*: a reliable crash-to-bug map would immediately make the fuzzing literature comparable, which would in turn let us find out which of the last decade's several thousand fuzzing papers described real improvements. I built a clustering tool for an internal pipeline once, over a fortnight I did not really have, and it worked. Nobody switched to it. Using it would have meant restating the previous quarter's numbers, and the pre-

[2] They are not standard largely because nobody's pipeline has a place to put them, which is a sentence that explains a great deal about our tooling.

vious quarter's numbers had already been presented to people who had presented them to other people. I have never worked out whether that was a tooling problem.

9.6 Problem 5: what the agents do

Since 2024, language-model agents have been finding real memory-safety defects in serious software — SQLITE, OPENSSL, the kernel — including bugs that had survived years of fuzzing and manual review, and in at least one case a bug its finder described as known to attackers before it was known to defenders.

Then take the clearest number available, because it is more startling than any general claim about the trend. Of the fourteen advisories carrying a 2025 identifier on OPENSSL's own vulnerability list, *thirteen* credit a single firm running automated analysis; the project's own release note for the January 2026 batch records twelve issues fixed, all disclosed by that one source. Count instead by the calendar year in which advisories were published and the ratio is three of four, because most of those thirteen appeared in January 2026 under 2025 numbers — a distinction worth making carefully, since the two counts answer different questions and only one of them is usually quoted. Either way, on one of the most heavily audited codebases in existence, the majority of a year's reported vulnerabilities came from a source that did not exist when the second edition of this book went to press. By mid-2026 the same list credits automated and machine-assisted discovery from more than one organisation, which matters more than any single firm's record: convergent results from competitors are evidence about the method, where a monopoly of one tool is evidence only about the tool.

This is the fastest-moving thing in the field and the honest position is that we do not know what it does to the equilibrium.

The optimistic reading: discovery is a defender-favouring capability, because defenders can run it continuously over code they own before shipping, and the marginal bug found before release costs an attacker an asset. The pessimistic reading: discovery scales for whoever has compute, the defensive side has to fix everything while the offensive side needs one, and Chapter 1's asymmetry is not cancelled by making search cheaper but amplified, because it now operates at machine speed. Respectable people hold both readings, and neither side has yet produced the measurement that would settle it.

What I will say to candidates is narrower and I believe it. The parts of this book that automation is already good at are the parts I would least mind losing: bulk triage, minimisation, variant sweeps, harness scaffolding. The parts it is worst at are attack-surface judgement, threat-model construction, calibration, and the sentence in §8.3. Those are the parts I have spent nine chapters arguing constitute the profession. That may be self-serving. Check it against what you observe over the next five years rather than against what your instructor found comforting to say in 2024. *Worth*: not a paper. An answer to the question of what this field is for in a decade.

NOTES AND SOURCES

Problem 9.2's practical cost is documented in [19].

Problem 9.3 needs a correction to earlier printings, which said that nothing better exists. That is not so, and Problems 9.2 and 9.3 are in any case one problem seen from two sides rather than two. The species-discovery framing [7] and the residual-risk estimators built on it [8] attack exactly this question and give sound upper bounds under stated assumptions. What remains unavailable is a *count* of the defects that remain, and the obstacle is the one this chapter names: a greybox campaign correlates its own samples, so

the independence those estimators want does not hold, and the same authors say so plainly. The honest position is that residual *risk* over the sampled space can be bounded, while the size of the sampled space is the quantity Problem 9.2 says we cannot measure.

Problem 9.5's figures are [28]'s.

Problem 9.6 is the fastest-moving material in this book and the most likely to have dated by the time you read it. The individual cases are [22] and [24], both from 2025. The counts are taken from OPENSSL's own published vulnerability list [34], read in July 2026, not from any vendor's announcement: fourteen advisories carry a **CVE-2025** identifier, of which thirteen credit Aisle Research, and the project's release note for 27 January 2026 states that all twelve issues in that batch came from that source. Two caveats belong with the number. It counts by identifier year; ten of the thirteen were published in January 2026, so counting by publication year gives three of four instead. And it counts the advisories listed for the 3.5 branch, so it is a count *on that list* rather than a cross-branch total. By mid-2026 the same list credits machine-assisted discovery from more than one organisation. That is why the text draws the distinction between convergent results and a single vendor's record.

Any general claim about proportions should be treated as provisional and checked against the current record — including this one, which is a snapshot of a list that changes several times a year.

9.7 Commencement

That is the ledger. Five debts, honestly stated, in a book otherwise made of things that hold.

Let me close where I opened. You came here because you noticed a gap between what a program was supposed to do and what it did, and you found the gap more interesting than the program. This book has spent nine chapters converting that noticing into apparatus: boundaries, guards, constraint chains, harnesses, envelopes, arguments. If the conversion feels like a loss, you have half understood it, so let me state the other half plainly. The method does not diminish the noticing; it funds it. Every guard you trace in Chapter 3 is a report somebody acts on. Every honest impact section in Chapter 8 is a patch that ships in the right order. The concave curve in Chapter 6 is the shape of a career long enough to train the people who replace you, and the tables in Chapter 7 are dry on purpose. The topology in Chapter 5 is Yelena Sorokin's, argued into me over four years, and she would have been pleased to hear a candidate call it obvious.

Carry these. *Be the person who says "I don't know"*, because in a field whose output is other people's decisions, calibration is the entire product, and it is astonishing how rare and how valuable it is. And *be kind to maintainers*. The bug you found has been sitting there for eleven years in code somebody wrote for free, and the fifteen minutes you spend making your report easy to act on will do more for the security of the world than the three days you spent finding it.

The software has not got safer on its own; the instruments have got very much better, and this term they are yours.

— J. A.

CHAPTER A

Toolchain Refresher

Written for the engineer who last wrote C some years and two frameworks ago. No apology is expected and none is owed; the machine does not care when you left, only that you have come back.

A.1 Five facts about memory

A pointer is a capability, not a number. It carries, in the language's model, an object it was derived from and a valid range. Arithmetic that leaves that range is undefined even if you never dereference the result, and that is why "I only computed the address" is not a defence and why compilers optimise on the assumption you did not.

Size is decided once and must be honoured everywhere. Every buffer has one size, and every access needs a bound derived from *that* size and not from a parallel computation. The sixth shape in §3.3 is what forgetting this costs.

Lifetime is a property of storage, not of pointers. Freeing invalidates the storage and not the copies of the pointer you kept, so use-after-free is a bookkeeping failure rather than a memory failure, and is found by reasoning about ownership.

Integer types promote, convert, and wrap according to rules nobody remembers. When it matters — and in a length computation it always matters — write down the types of every operand and the type of the result. The bug is usually visible once you have.

Adjacency is not part of the language and is part of the attack. Nothing in C tells you what follows your allocation; everything about the exploitability of your overflow depends on it. Hold both facts at once.

A.2 The debugger in nine commands

Fluency here is worth more than any tool you will install this year. **break f**, **run**, **bt** — where am I and how did I get here. **p expr**, **x/16xb addr** — what is the value, what are the bytes. **watch *addr** — tell me when this changes; the single most useful command in corruption work. **finish**, **until** — get out of here. And with a recording debugger, **reverse-continue**: run backwards to the write that broke it. Learn that one early. It converts the characteristic two-day investigation of §8.3 into an afternoon, and candidates who postpone it postpone it for years.

A.3 The flags that matter

-g always; **-O1** for fuzzing (**-O0** hides optimiser-dependent bugs, **-O2** makes stacks harder to read); **-fno-omit-frame-pointer** for usable traces. **-fsanitize=address,undefined** as the default oracle; **-fsanitize=fuzzer** for a libFuzzer harness. **-D_FORTIFY_SOURCE=3** where your toolchain supports it (GCC 12 and **glibc** 2.35 onwards; **=2** otherwise), **-fstack-protector-strong**, **-Wl,-z,relro,-z,now** when you want to know what the shipped binary's mitigations are — and **checksec** or equivalent to find out what they actually were, because the answer is regularly not what the build system claims. **-fsanitize-coverage=trace-cmp** to help the fuzzer up comparisons.

A.4 Reading a stack trace

Read it outwards, not downwards. Frame 0 is where the program noticed; the frame you want is usually the outermost one still inside the component you are auditing, because that is where the attacker's data entered code the maintainers own. Ignore the allocator frames. Note the frame where a size or length was last computed — that is your candidate bug site. And if the trace is implausibly short or contains addresses without symbols, you are looking at a corrupted stack, which is itself a finding about the class of bug.

A.5 The discovery curve

The formulas that carry Chapter 4 are worth being able to reproduce. Over N independent executions with per-execution trigger probability p,

$$\Pr[\text{at least one trigger}] = 1 - (1-p)^N \approx 1 - e^{-pN}, \qquad \mathbb{E}[\text{time}] = \frac{1}{pr}.$$

The first gives you confidence from a completed campaign; the second gives you a budget before one. Both are wrong in the same direction — executions are not independent, because the fuzzer deliberately correlates them — and both are still the right first estimate, which is the ordinary condition of applied arithmetic. Use them, quote them with the assumption stated, and remember Problem 9.3: nobody has anything better.

CHAPTER B

Tables, Protocols, and Answers

B.1 Choosing an oracle

Instrument	Cost	Finds / misses
ASan	1.73× published; 1.6–2.7× measured	heap/stack/global overflow, UAF, double free / uninit reads, non-linear overflows, intra-object
UBSan	1.02–1.04× measured	undefined arithmetic, bad shifts, misaligned and null pointers / anything defined-but-wrong
MSan	3× documented, full-program build	uninitialised reads / everything else
TSan	5–15× documented	data races / non-concurrent bugs
LeakSan	at exit only	leaks / correctness
Valgrind	6–14× measured, no rebuild	most memory errors on binaries you cannot compile
KASAN + **kcov**	kernel build	kernel memory errors, with coverage
Assertions	none	invariant violations, the cheapest oracle in existence and the most neglected

Read that column with more suspicion than a table invites. “Published” and “documented” are the figures the tool’s own authors report, on their benchmarks; “measured” is this book’s, on a parser-shaped workload of the kind you are likely to instrument. They disagree, and the disagreement is not error — *a sanitiser’s slowdown is a ratio, and a ratio shrinks as the work underneath it gets more expensive*. Changing nothing but the type of an accumulator in the benchmark moved ASan from 2.6× to 1.6× and Valgrind from 14× to 5.8×. Turning on Valgrind’s origin tracking moved it back up to 22.7×. So a single number for a tool is a number for somebody’s benchmark, and the useful figure is always the one you measure on your own target — which is what Exercise 2-5 is for. Note also that Valgrind is not commensurable with the rest: it instruments a finished binary at run time rather than being compiled in. That is why it is the entry for code you cannot rebuild.

B.2 Bug class to method

Class	Best finder	Note
Linear heap overflow	fuzz + ASan	the easy case; mostly already found
Non-linear OOB write	reading	ASan is blind to it (§2.5)
Integer overflow in size	reading + UBSan	§3.3(i); grep for *** sizeof**
Signedness confusion	reading	§3.3(iii); grep for signed **len**
Off-by-one at terminator	reading	ask: does the count include the NUL?
Use-after-free	fuzz + ASan	quarantine must not be exhausted
Type confusion	reading + variant analysis	§3.6; the Ghostscript pattern
Uninitialised read	MSan	invisible to everything else
Data race	TSan + stress	under-hunted relative to severity
Parser differential	differential fuzzing	§5.4; does not crash
Logic / authorisation	reading, with a threat model	no tool models your boundary

B.3 Campaign checklist

Before starting: reconnaissance done (§6.2); surface chosen and written down; harness contract stated; determinism verified by running one input twice;[1] input size capped; leaks clean; seeds gathered and minimised; dictionary written; build matrix configured; output on local disk with a size limit; network disabled; container isolated.

Weekly: re-minimise corpus; regenerate and *read* the coverage report; check the covered set includes what you care about; check the encoder as well as the decoder; rotate power schedules; review whether *m* still looks right (§6.2).

Never: report a crash count; conclude absence from a plateau; run the unsanitised build only; fuzz something you do not own.

[1] Enumerate your sources of entropy before the campaign rather than after the crash you cannot reproduce: the clock, the address-space layout, the thread scheduler, uninitialised memory, the network, the filesystem's iteration order — and, in one instance on file here, a cat who crossed a keyboard during trial six of ten and cost the candidate a week. Entropy is whatever you failed to enumerate.

B.4 Proof-of-concept bundle

1. minimised input, as a file, with length and hash
2. target commit hash
3. exact build command, including compiler version and sanitiser flags
4. exact run command, one line
5. full unedited sanitiser output
6. **Dockerfile** or equivalent reproducing from a clean base
7. root cause as the §8.3 sentence, with file and line
8. impact as the three-premise argument, with uncertainties stated
9. regression test in the project's own format
10. disclosure timeline and credit preference

B.5 First-contact report template

```
Subject: Security: heap out-of-bounds write in parse_table() (untrusted
    input)

Hello,

I found a memory-safety defect in <project> and would like to report it
privately. I am following a 90-day coordinated disclosure timeline from
today (<date>), with a 14-day grace period if a fix is scheduled.

Summary:  Heap out-of-bounds write of 1 byte, attacker-chosen value, in
          parse_table() (src/table.c:412), reachable by parsing any
          untrusted input file. No privileges or non-default options
          required.
Affected: <versions>; introduced in <commit> (<date>).
Attached: minimised 9-byte reproducer, Dockerfile, full ASan output,
          and a regression test against your tests/ layout.

Root cause: at src/table.c:412 the destination size is computed from
n_entries under the assumption that the header count matches the number
of entries actually written, which does not hold when the count field
is smaller than the entry list, because the writer loop is bounded by
the list rather than by the count.

Impact: I have characterised the primitive (1 byte, chosen value,
controlled offset within a 64-byte range, repeatable) but I have NOT
assessed exploitability and make no claim about it.

Happy to help test a fix, and to delay publication if you need more
time -- just tell me what would help. Please let me know if you would
prefer I request the CVE or if you will assign it.

<name>, <affiliation>
```

Note the four things this template does that candidates omit: it states the timeline in the first paragraph rather than the last; it separates the primitive from an exploitability claim it does not make; it offers flexibility without surrendering the deadline; and it asks who should request the identifier instead of assuming.

B.6 Solutions

Answers to every unstarred exercise. Starred exercises are supervised lab work whose output is a measurement, a rig or a document rather than a result that can be printed; for those a marking rubric is given instead, naming what a complete submission must contain.

Chapter 1

1-1. The code is identical; the trust boundary is not, and the boundary is what the exercise is asking about. (a) The boundary encloses the user and their own files: the actor supplying input is the actor who owns the machine, so a defect is a robustness bug rather than a vulnerability, and κ is essentially nil. (b) The gateway processes attachments from anyone on the internet; the reachable interface is unauthenticated and remote, and the consequence class rises to code execution in a service context. (c) The library sits inside a renderer that already treats page content as hostile, and the boundary it must hold is the one between a web page and everything else on the machine. The vendor's answer differs because the vendor's threat model differs; a report that does not state which of the three you mean is answerable with "works as intended" in all three cases.

1-2. *Tested:* "Over 168 hours on one harness reaching $X\%$ of lines, no crashes were observed under ASan and UBSan." *Not tested:* "This says nothing about code the harness cannot reach, about bug classes without an oracle in this build, or about inputs outside the harness's input model." *The question:* "What was the coverage, and which subsystems were not in it?" If your colleague cannot answer, the week bought an unknown quantity. Compare with Theorem 5.1 after §5.1.

1-3. (a) Defect; needs a release build to become reachable. (b) Reachable defect; needs an availability impact argument to become a finding. (c) Controlled reachable, low χ. (d) Controlled reachable, higher χ; the value is what separates them. (e) Reachable, and its consequence class is disclosure rather than corruption. That is a different report, and not a lesser one.

1-4. No fixed answer; the habit is the point. A complete answer names, among the "not previously thought of" entries, at least one of: file names and paths, environment variables, terminal escape sequences in anything the program prints, clipboard contents, image or document metadata, or a configuration file writable by a less privileged user. The distinguishing question for each is not "does data arrive here" but "can somebody who is not me choose it".

1-5. No fixed answer. A complete answer states two beliefs and their holders in the form of §1.1: for instance, "the allocation site believed the header's length field bounded

the payload; the copy loop believed the payload bounded itself". The second paragraph identifies what the finder saw that you did not, usually a reachability condition rather than the defect itself.

Chapter 2

2-1. The check is present in all four builds of the first listing on both compilers, because neither can prove $b \geq 0$. In the second listing, where **b** is unsigned, the branch is eliminated at **-O1** and above on both. The correct rewrite tests the operands rather than the result, **if (b > INT_MAX - a)** for the signed case, or uses **__builtin_add_overflow**, and survives every optimisation level because it does not depend on a value the standard says cannot exist. The one-sentence answer: reading tells you what the source says, and the only thing that tells you what will execute is the thing that executed.

2-2. "Only padding" is a statement about one allocation order in one run. Two things an attacker can vary: the *sequence* of allocations before the overflowing one, which decides which object lands adjacent, and the *size* of requested objects, which decides the size class and therefore the neighbourhood. A third, on some allocators, is timing against a freelist. The property of the bug is the write; the identity of the victim is a property of the run.

2-3. (a) ASan. (b) Nothing reliably; this is the non-linear case, where the index steps clean over the redzone. (c) Nothing; the reuse is silent. (d) MSan. (e) UBSan. (f) TSan. The two undetected classes are the two that produce the longest-lived real vulnerabilities, which is not a coincidence: an oracle that does not fire is a bug that does not get reported.

2-4. The overflow stays inside one allocation, so the shadow map says "addressable" throughout and nothing fires. The change to the struct is to separate the array from its neighbours: give it its own allocation, or insert explicit padding that is never read. Maintainers decline because it costs memory and cache locality on every instance of the struct to detect a bug class they do not believe they have, and because the change is invasive in a way the fix for an actual reported overflow is not. That refusal is not unreasonable; it is the trade the §2.5 discussion is about.

2-5*. *Rubric.* A complete submission reports executions per second for each of the three builds on an identical corpus, states the machine and compiler versions, and gives the ratio rather than only the raw numbers. It must note that the ratio depends on the workload, since allocation-heavy code pays far more than compute-heavy code, and it must not present its measurement as a general figure for the tool. Credit for observing that UndefinedBehaviorSanitizer's cost depends on how many checked operations the target performs, and is close to free on code that is mostly unsigned arithmetic.

Chapter 3

3-1. Three fixes, and the order is load-bearing. *First*, verify four header bytes remain, **in_len - off >= 4**, before reading them or computing anything from them. *Second*, bound **len** against **sizeof(name) - 1** *and* against the remaining input **in_len - off - 4**. *Third*, make **off**, **len**, and **in_len size_t**. The invariant to write down is: *at every loop iteration, off* $+ 4 +$ *len* $\leq$ *in_len and len* $<$ *sizeof(name).*

Now notice why the order matters, because this is §3.3(iv) waiting for you in your own patch. Once every variable is **size_t**, **in_len - off - 4** is an *unsigned* subtraction. Compute it before establishing that four bytes remain and it wraps to something near 2^{64}, the bound it exists to impose evaporates, and two of the three original defects survive a fix that looks complete. A guard that establishes nothing is the first failure mode in the proof of Theorem 3.2. Apply the fixes in the order above and the parser is clean; apply them in the order a hurried reader would, and it is not.

3-2. (i) Multiplication overflow in an allocation size: detected by UBSan's default set only if the operands are signed. In **size_t** it is defined wraparound, so the default set is silent and only the opt-in **unsigned-integer-overflow** check reports it (§2.5). That is why that check exists and why it is off by default. (ii) Truncation across widths: **no sanitiser**, because narrowing conversions are well-defined. (iii) Signedness confusion: UBSan catches the negative-index access if one occurs, but not the conversion itself. (iv) Unsigned subtraction below zero: defined behaviour, so nothing fires until the huge result is used as a size, at which point ASan may see the resulting access. (v) Off-by-one at the terminator: ASan, if and only if the object has a redzone next to it. (vi) The size decided once and used twice: **no sanitiser**, because every individual operation is well-defined; only the disagreement is wrong. The two undetectable shapes are (ii) and (vi), and they are the two that require a human to notice that two computations of the same quantity disagree.

3-3. The invariant **kTableSize** was meant to guarantee is: *the allocation is large enough for every table the code will build from this input*, first-level and second-level together. The assertion to add at the allocation site compares the size actually consumed by table construction against the size allocated. In practice, have **BuildHuffmanTable** take the buffer length and assert it never writes past it, the form the upstream fix took.

3-4. No fixed answer. A complete plan names, for each excluded surface, the reason for exclusion: not reachable by the threat model, already covered by an existing campaign, requires privileges the attacker does not have, or too expensive to reach in the time. A plan that excludes nothing has not been written; a plan that excludes without justification is not defensible to the client afterwards.

3-5*. *Rubric.* A complete submission contains the initial over-specific query and evidence it matched the pre-patch code and not the post-patch code; three successive generalisations, each with the match count and the true-positive count after triage; and the plotted curves. Full credit requires a sentence on the shape of the gap, that precision falls faster than recall rises, and an explicit statement of where the submitter would have stopped generalising in a real engagement, with the reason.

Chapter 4

4-1. $p_{\text{blind}} = 2^{-(32+8)} \cdot (8/2^{16}) = 2^{-53}$, where the magic and the version are equalities, the record count is a range constraint admitting 8 of 2^{16} values, and the 32-bit magic dominates. Blind expected time at 5×10^4 exec/s is around 5,700 years. Guided work is $2^{32} + 2^8 + 2^{13}$, i.e. about a day at the same rate, and CmpLog reduces the 2^{32} term to approximately nothing. That is why the honest answer to this exercise is "minutes, with the right flags". Note the trap in the arithmetic: the count field must be charged once, either inside the exponent or as an acceptance ratio, never both.

4-2. $N = 2 \times 10^4 \times 8 \times 1.728 \times 10^5 \approx 2.8 \times 10^{10}$; with no find, $p \lesssim 1/N \approx 3.6 \times 10^{-11}$ at roughly 63% confidence (at $pN = 1$ the probability of seeing nothing is e^{-1}), and one order of magnitude looser for comfortable confidence. The two sentences: this establishes an upper bound on the trigger probability of any defect *reachable by this harness*; it establishes nothing whatever about defects outside the covered set, which is Theorem 5.1 and Problem 9.2.

4-3. Doubling the cores doubles N and therefore halves the expected time: a linear factor. A dictionary supplies the constant side of an equality constraint, removing a term from the exponent of Principle 4.1 rather than scaling the coefficient. One beats the other by orders of magnitude whenever a constraint is actually blocking. The opposite case: when no constraint is blocking, meaning the corpus already reaches the interesting code and the bug simply has a low per-execution probability, there is no exponent left to remove, and the linear factor is all there is to buy.

4-4. In order: (1) generate a coverage report and read it; observation: is the subsystem you care about in the covered set at all? (2) Check for a checksum or magic rejecting inputs early; observation: does patching it out move coverage? (3) Add entry points, including the encoder; observation: new edges in previously cold files. (4) Move to structure-aware generation; observation: the fraction of executions surviving past the parser front end. (5) Strengthen the oracle with sanitisers or assertions; observation: crashes on inputs already in the corpus. (6) Only now, more time or cores, justified by an explicit estimate of p and the N required, per §4.3, and by nothing else.

4-5. "The campaign produced 1,200 crashing inputs, not yet triaged." The smallest

work that replaces it with something meaningful: minimise all of them mechanically, bucket by error class and top frames as a work-ordering device, then root-cause one representative per bucket until the buckets stop yielding new causes. The number that means something is the count of distinct root causes, and it is usually smaller by two orders of magnitude.

4-6*. *Rubric.* A complete submission reports ten trials per configuration with the full distribution, not the best run; a Mann–Whitney U test between configurations; the Vargha–Delaney $\hat{A}_{12}$ effect size with its interpretation; and time to first crash where one occurred. Full credit requires acknowledging that ten trials is fewer than the reference literature uses, and stating what that costs in confidence. A submission that reports only means, or only one trial, receives no credit regardless of the numbers.

Chapter 5

5-1. In rough order of cost to distinguish: (1) the harness never calls the code: read the coverage report. (2) An early **return** on an unmet precondition: look at the first uncovered line in each cold function. (3) A checksum or magic rejecting inputs: patch it out in a test build and compare. (4) The code is reachable only from a different entry point: read the callers. (5) The code is genuinely dead in this configuration: check the build flags. Each is cheaper to test than the one after it, and that ordering is what the exercise wants.

5-2. (a) Stateless transformer: one **LLVMFuzzerTestOneInput** over the decode entry point. (b) Stateful protocol: fuzz the record layer in-process, not over a socket. (c) Stateful API: interpret the input as a sequence of allocator operations. (d) Stateless transformer, but with a structure-aware input model, since ELF has interdependent offsets. (e) Kernel interface: a structure-aware system-call generator with resource types, in a disposable virtual machine. (f) Differential: two implementations of the same normalisation, compared. The interesting bug is disagreement rather than a crash.

5-3. *Uninitialised **out**:* the false finding is a crash inside **img_decode** reading a field no real caller leaves unset. You cannot recognise it from the ASan report, because ASan does not detect uninitialised reads at all, which is itself the lesson; you need MSan, or you need to notice that the allocation stack for the object is absent because there is no allocation. *Unbounded **n**:* the report shows an allocation size or index far larger than any real caller could request; the giveaway is that the size in the report has no plausible provenance in the input format. *The leak:* LeakSanitizer names an allocation whose free is absent from the harness rather than from the library, and the allocation stack terminates in your own code.

5-4. Compare the decoded content, not the bytes: the set of member names, their con-

tents, their sizes and their metadata, each normalised: order canonicalised, timestamps zeroed, path separators unified, character encodings settled. Three classes to ignore deliberately: differences in fields the format leaves implementation-defined; differences in error *messages* where both implementations reject the input; and differences in resource limits where one implementation declines an input the other accepts. The ignore list is the exercise because a differential harness without one reports the specification's own ambiguity as a finding.

5-5. A dataflow query abstracts away path conditions, so it reports paths no input can take: its errors are false positives. A symbolic executor constructs a real input for every path it reports, so it cannot report a path that does not exist; its errors are the paths it never reaches, which are false negatives. Use both when you have a large codebase and a specific dangerous sink: the query enumerates candidate paths cheaply, and the executor is pointed at the handful that matter to decide which are real.

5-6*. *Rubric.* A complete submission reports edge coverage for both configurations across ten trials, and a caveat of the form: *this campaign establishes coverage and absence of crashes for the code path reachable with checksum verification disabled; it establishes nothing about the shipped binary, in which no input reaching this code can be constructed without a valid checksum.* Credit depends on the caveat.

Chapter 6

6-1. $4m/(1+m)^2$ at $m = 3$ and $m = 1/3$ is 0.75 in both cases. The 3 a.m. sentence: *you are throwing away a quarter of everything, and the quarter is coming out of you.* At $m = 3$: building a structure-aware protobuf harness for a library whose bugs are off-by-ones in error paths. At $m = 1/3$: grepping for **strcpy** in a codebase whose defects are state-machine confusions across a protocol handshake.

6-2. (a) $m \approx 0$: the existing harnesses define the covered set, and four years of campaign has exhausted what it can reach. (b) m high: the encoder is a surface the existing harnesses do not touch at all. (c) m high: the project's own fixed bugs are evidence of the mistakes its authors habitually make. (d) m moderate to high: error paths are undercovered by construction, since a fuzzer reaches them only by accident. Ranking: (b) $\approx$ (c) $>$ (d) $>$ (a). The justification is §5.1, that the covered set bounds what a campaign can find, and not a preference for reading over running.

6-3. *(a)* With $r(F) = F/\tau_r$, equilibrium is $F^* = kP_0\tau_r$, which sits below F_c exactly when $P_0 < F_c/(k\tau_r)$, a load ceiling that scales with recovery time. With $r(F) = r_{\max}\tanh(F/F_s)$, equilibrium requires $kP_0 = r_{\max}\tanh(F^*/F_s)$, which has no solution once $kP_0 > r_{\max}$; hence $P^* = r_{\max}/k$, and below it $F^* = F_s\,\mathrm{artanh}(kP_0/r_{\max})$. Differentiating, $dF^*/dP_0 = kF_s/(r_{\max}(1-(kP_0/r_{\max})^2))$, which diverges as $P_0 \to P^*$.

(b) Near the ceiling, a small increase in load produces a very large increase in the fatigue you eventually settle at. Expect to feel the load increase. Expect *not* to feel the change in equilibrium, because it is a property of the steady state rather than of the moment and it arrives on a delay, so the felt cost of the last increment is a poor guide to its real cost. Full credit for the second list; the first is arithmetic.

(c) No fixed answer, and the grading is on the obstacles rather than the design. A defensible sketch: practising researchers rather than undergraduates, weekly deep-work hours as the manipulated variable, an objective task-performance measure rather than self-report, months rather than days, and enough subjects to detect an effect in a population that self-selects for stamina. Obstacles that are not funding, at least one of which should appear: you may not ethically drive subjects towards F_c, which is the region the model is entirely about; "intensity of sustained deep attention" has no agreed operational definition, so P is not a measurable quantity; the outcome that matters is career-scale and the study is not; and the population is small, employed, and largely unwilling to be logged. A candidate who answers "nobody has funded it" and stops has missed the exercise.

(d) Van Dongen et al. found the dissociation of (b) under chronic sleep restriction: subjective sleepiness responded once and then flattened while objective deficits went on accumulating for fourteen days. That supports the claim that the warning sign is absent. It does not support the functional form, the existence of P^*, or any transfer to code reading: the load variable was sleep rather than attention, the task was psychomotor vigilance, the subjects were not researchers, and fourteen days is not a career. A candidate who says the model is "confirmed" has failed the part of the exercise the rest of it exists for.

6-4. Strict concavity with $H(0) = 0$ gives $H(\Delta) > \frac{1}{2}H(2\Delta)$, hence $2H(\Delta) > H(2\Delta)$. For $H(\Delta) = H_0\Delta/(1+\Delta/\Delta_0)$, the gain is $2H(\Delta)/H(2\Delta) = (1+2\Delta/\Delta_0)/(1+\Delta/\Delta_0)$, which at $\Delta = \Delta_0$ is $3/2$. Two people at half the duration deliver half again as much.

6-5. A complete checklist contains, at minimum: *coverage*: what is the covered-set delta since the last review, and did any subsystem lose coverage? *Diff*: what entry points changed, were added or were renamed since the last review, and does a harness still reach each? Plus: is the encoder covered; is the corpus still minimised; does each harness still build in every configuration in the matrix; and who is the named owner. A checklist without a named owner is not a control because nobody is accountable for its not having been run.

6-6*. *Rubric.* Four weeks of session-level records, findings attributed to the activity that produced them, and a realised yield per hour by category. Full credit requires the submitter to state where the distribution differs from their prior expectation and what

they intend to change, since the exercise's purpose is calibration rather than measurement.

Chapter 7

7-1. Pattern III, scoped hard, or Pattern I if the five days include the reporting. Reconnaissance says $m \approx 0$ for byte mutation on the existing harnesses, so the plan does *not* run the existing fuzzers. It does: variant analysis over the project's own fixed bugs; a harness for the encoder or another untouched entry point; and manual reading of error paths. What you will not do: re-fuzz the decoder, attempt whole-codebase coverage, or promise a finding.

7-2. Coverage delta since last review; entry points added or changed in the diff and whether a harness reaches each; determinism re-verified by running one corpus input twice in each build; encoder coverage explicitly reported. Two sentences on ownership: a checklist with no named owner has no one to notice it was skipped, and a control that can be skipped without anyone noticing is documentation rather than a control.

7-3. The reply must grant the client's underlying interest, since they want assurance rather than a document, and then explain that a boundary document is what makes the assurance meaningful: without a stated scope the report cannot distinguish "we looked and found nothing" from "we did not look", which is the distinction they are paying for. Offer the compromise that makes it acceptable: the boundary document names the surfaces in priority order, and time remaining after the priority surfaces is spent down the list, so nothing is excluded permanently, only ordered.

7-4. With the documented two-to-five-times penalty, take the pessimistic end: $N \approx 3.3 \times 10^{10}/5 \approx 6.6 \times 10^{9}$. At $p = 10^{-9}$, $pN \approx 6.6$ and detection is around 99%; at $p = 10^{-10}$, $pN \approx 0.66$ and detection is around 49%, so 10^{-10} is the marginal value and roughly an order of magnitude of bug is lost relative to the source build. The two recoveries are a structure-aware input model and a dictionary recovered by reversing. Neither is optional because neither can be substituted by hardware: both change the exponent, and cores change only the coefficient.

7-5. The ordering is the exercise. Stop the campaign. Do not copy, move or open the data. Record the time and the artefact identifier. Notify the named engagement contact and your own supervisor, in that order, by the channel the scope document specifies. Preserve the artefact in place under existing access controls; do not delete it, because deletion destroys evidence and is itself a reportable act. Await written instruction before any further handling. Everything else, meaning notification of data subjects, regulatory reporting, and whether the engagement continues, is the client's decision and not yours.

7-6*. *Rubric.* A working rig, demonstrated: a kernel built with coverage and sani-

tiser instrumentation, a disposable virtual machine that can be restored from snapshot, and a structure-aware system-call generator reaching a chosen driver with evidence of coverage in it. The submission is the environment, not any finding; a crash during the demonstration is neither required nor disqualifying.

Chapter 8

8-1. A correct implementation partitions the input into n subsets, tests the complements, recurses on any that still fails, doubles n when nothing does, and stops when n exceeds the input length. The result is 1-minimal by construction: at the final granularity every single element has been removed in turn and the failure disappeared each time. An input on which 1-minimal is not globally minimal: let the predicate fail if the input is exactly $\{1,\ldots,8\}$ *or* exactly $\{3,7\}$, and start from $[1,\ldots,8]$. Removing any single element breaks the first condition without reaching the second, so the algorithm returns all eight, yet $\{3,7\}$ also fails and is a quarter the size. This is acceptable because 1-minimality is a statement about single removals, and searching all subsets is exponential; the guarantee you are buying is cheapness, not optimality.

8-2. In order: the allocation stack, which tells you where the size was decided; the ASan report gives it free. The size actually allocated, compared with the size the format implies, which usually is the bug. Whether the offset is fixed or input-dependent, established by varying the input and re-running. Whether the written value is attacker-chosen, established the same way. What object occupies the adjacent memory in this build, established under a debugger. And whether the write can be repeated without crashing, established by construction. Only then is there enough to write an impact section.

8-3. (a) "A one-byte out-of-bounds heap write with attacker-chosen value at an offset controlled within *range*; exploitability not assessed." (b) "*n* crashing inputs; after minimisation and triage, *k* distinct root causes." (c) "Allocation order at the overflow site is influenced by input via *this mechanism*; the adjacent object in our test build was *X*." (d) "Availability impact confirmed; memory-safety violation present, so absence of further impact is not established", because a crash is a floor on severity, never a ceiling.

8-4. No fixed answer. The measurable criterion is in the exercise: if the reader asks a question, the report failed. The usual causes are an unstated compiler version, a build that assumes a tool the reader does not have, an input pasted rather than attached, and a root-cause paragraph that describes the crash rather than the defect.

8-5. No fixed answer. A complete policy states a deadline, a grace period, an in-the-wild track, a position on publication and a position on bounty terms that restrict it. The rule most people find hardest to keep is the deadline, in the specific case where the vendor is responsive, apologetic and simply slow, which is the case the policy exists for,

because the unresponsive vendor is easy.

8-6*. *Rubric.* An independently reconstructed reproducer, a root cause in the sentence form of §8.3, an impact argument written before reading the advisory, and a one-page comparison identifying what was missed and in which direction the submitter was miscalibrated. The comparison is the deliverable; the reproducer is only evidence that the comparison is honest.

Glossary

Terms as this book uses them. Where a term is contested or used differently elsewhere, the entry says so — an unmarked disagreement with the literature is how a reader ends up confidently wrong in a meeting.

Attack surface ($\mathscr{S}$)
The set of interfaces across which an actor outside the trust boundary can supply input. Enumerating it is the first step of §3.5 and the habit Exercise 1-4 exists to build.

Concolic execution
Symbolic execution guided by a concrete run, so that path conditions are collected along a path known to be feasible. Under-approximates: what it returns is real.

Constraint chain
The sequence of conditions an input must satisfy to reach a given program point. Blind search pays the product of their probabilities; coverage-guided search pays approximately the sum (Principle 4.1).

Corpus (Q)
The set of inputs a fuzzer retains because each reached something no earlier input reached. A corpus is a record of solved constraints. That is why seeding matters and why minimising it periodically matters more than candidates expect.

Coverage
The set of program locations or edges executed. A computable proxy for progress, a poor target in itself, and the only honest basis for the sentence "we did not look there".

Differential fuzzing
Running two implementations of one specification on the same input and treating disagreement, rather than corruption, as the oracle. Finds the bug class that does not crash (§5.4).

Fidelity (ϕ)
The degree to which states a harness can construct correspond to states an attacker can reach. Defined in §5.3, used throughout, and computable by nobody — which is Problem 9.2.

Guard
A check that establishes a sink's precondition *on the specific value that reaches it*. The qualifier carries Theorem 3.2.

Harness ($\mathscr{H}$)
A program mapping a byte string to a sequence of calls into the target, together with the build configuration those calls run under. The build is part of the harness; two harnesses differing only in sanitiser flags are different harnesses.

Input-to-state substitution
Instrumenting comparisons so the values they actually see can be spliced back into the input at matching offsets, collapsing equality constraints that blind mutation would never satisfy. Implemented as CmpLog in AFL++ and natively in libFuzzer; the reason Figure 4-1 looks as it does.

Intra-object overflow
A write that passes the end of one struct member into another within the same allocation. In bounds and in lifetime as far as an allocator-based tool can see, and therefore invisible to AddressSanitizer's default configuration.

Minimisation
Reducing a failing input until every remaining element is load-bearing. The standard algorithm is *ddmin*, whose result is 1-minimal — no *single* element can be removed — which is weaker than globally minimal and cheap enough to be worth it.

Oracle
The mechanism that decides an execution misbehaved. Sanitisers, assertions, differential comparison and round-trip identity are all oracles; a crash is only the weakest of them, and the largest single determinant of a campaign's yield.

Primitive
What a defect gives an attacker, described in terms of offset, value, length and repeatability rather than in terms of an outcome. "One-byte write of a chosen value at a controlled offset" is a primitive; "remote code execution" is a claim about what someone could build from one.

Reachability (ρ)
Whether an input exists that drives execution to a program point, consistent with the interfaces an attacker actually reaches. In general a path condition rather than a yes or a no.

Redzone
Poisoned memory placed around an allocation so that adjacent accesses are detected.

Finite in size. That is why an access far enough past the end steps over it unnoticed.

Sanitiser (Σ)
Instrumentation plus a runtime that converts a class of silent undefined behaviour into a loud, immediate, attributable abort.

Sink
An operation with a non-trivial precondition on its operands: a write, an index, an allocation size, a free, a cast, a command.

Source
A program expression whose value is, wholly or partly, chosen by an actor outside the trust boundary.

Spatial safety
The property that an access falls entirely within the bounds of the object its pointer derives from.

Stack hash
A fingerprint of a crash's call stack, used to group crashing inputs. Both over- and under-counts distinct defects, so it orders work but never counts bugs (Problem 9.5).

Structure-aware mutation
Mutating a structured representation of the input and serialising it, rather than mutating bytes, so that checksums and length fields stay consistent by construction and every execution tests program logic instead of format validity.

Temporal safety
The property that an access occurs within the lifetime of the object it refers to.

Trust boundary ($\mathscr{T}$)
The line across which an actor is not trusted. Consequence is defined relative to it. That is why a boundary you did not write down is the one your finding will be closed against.

Type confusion
Access to an object through a pointer to an incompatible type: in bounds, in lifetime, and interpreted as the wrong thing.

Variant analysis
Abstracting a known defect into a pattern and sweeping the pattern across a codebase, and then across every codebase you can index. The cheapest bug you will ever find is the second instance of one you have already understood.

Bibliography

Every source here was read before it was cited. Where the text gives a figure, the figure comes from the source named and not from a summary of it; where a source and this book disagree, the source wins and the text has been changed. Tool documentation is cited with the version it was checked against, because flags move. Web sources carry an access date for the same reason.

Bibliography

[1] Aleph One. “Smashing the Stack for Fun and Profit.” *Phrack* 49, file 14, November 1996.

[2] Fioraldi, A., Maier, D., Eissfeldt, H., and Heuse, M. “AFL++: Combining Incremental Steps of Fuzzing Research.” *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[3] AFL++ project. *Fuzzing Binary-Only Targets.* AFL++ documentation, **aflplus.plus/docs**. Accessed July 2026. Source of the two-to-five-times figure for QEMU and FRIDA modes in §7.5.

[4] Arcuri, A., and Briand, L. “A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering.” *33rd International Conference on Software Engineering (ICSE)*, 2011.

[5] Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., and Holz, T. “REDQUEEN: Fuzzing with Input-to-State Correspondence.” *Network and Distributed System Security Symposium (NDSS)*, 2019.

[6] Bishop Fox. “An Exploration of JSON Interoperability Vulnerabilities.” Accessed July 2026. Forty-nine parsers tested; seven take the first key on duplicates.

[7] Böhme, M. “STADS: Software Testing as Species Discovery.” *ACM Transactions on Software Engineering and Methodology* 27(2), 2018.

[8] Böhme, M., Liyanage, D., and Wüstholz, V. “Estimating Residual Risk in Greybox Fuzzing.” *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[9] Brubaker, C., Jana, S., Ray, B., Khurshid, S., and Shmatikov, V. “Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations.” *IEEE Symposium on Security and Privacy*, 2014.

[10] Cadar, C., Dunbar, D., and Engler, D. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[11] Carnegie Mellon Software Engineering Institute. *SEI CERT C Coding Standard.* Rules INT30-C, INT32-C, MEM07-C, MEM35-C. Accessed July 2026. MEM07-C is the source for the **calloc** discussion in §3.3.

[12] CERT Coordination Center. *Vulnerability Disclosure Policy.* Carnegie Mellon University. Accessed July 2026.

[13] Cybersecurity and Infrastructure Security Agency. *Coordinated Vulnerability Disclosure Process.* Accessed July 2026.

[14] LLVM Project. *AddressSanitizer*, *MemorySanitizer*, *ThreadSanitizer*, *UndefinedBehaviorSanitizer* and *libFuzzer* documentation. Checked against Clang 21.1.8, July 2026.

[15] Cowan, C., Pu, C., Maier, D., et al. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." *7th USENIX Security Symposium*, 1998.

[16] Dowd, M., McDonald, J., and Schuh, J. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities.* Addison-Wesley, 2006.

[17] Drake, J. J. "Stagefright: Scary Code in the Heart of Android." Black Hat USA, 2015; and Zimperium zLabs advisories for CVE-2015-1538. Source for §3.3(i). The identifier covers an integer overflow in **SampleTable::setSampleToChunkParams** (**SampleTable.cpp**, **libstagefright**), reached by a crafted **stsc** atom and fixed in Android 5.1.1 LMY48I. The campaign that found it ran AFL at roughly 3,200 executions per second over more than six CPU-months.

[18] Cisco Talos. TALOS-2016-0095: "Lhasa LHA Level-3 Header Integer Underflow" (CVE-2016-2347). 2016. Source for §3.3(iv). The subtraction is **header_len - RAW_DATA_LEN(header)** with both operands unsigned and no lower bound enforced on **header_len**; affects Lhasa 0.0.7 and 0.3.0.

[19] GitHub Security Lab. "Bugs that Survive the Heat of Continuous Fuzzing." Accessed July 2026. Source for the decoder/encoder asymmetry cited in §4.6, §5.3, §6.6, §7.3 and §9.2.

[20] GitHub Security Lab. "Ghostscript Type Confusion: Using Variant Analysis to Find Vulnerabilities." Accessed July 2026. Six new type confusions from a query modelled on prior work; CVE-2018-19134, CVE-2018-19476, CVE-2018-19477.

[21] Google Security Blog. “Eliminating Memory Safety Vulnerabilities at the Source.” September 2024. Source for the 76% (2019) to 24% (2024) figures in §1.3.

[22] Google Project Zero and DeepMind. Big Sleep. CVE-2025-6965 (SQLite), July 2025. Characterised by Google as “known only to threat actors and at risk of being exploited”.

[23] Google Project Zero. “Policy and Disclosure: 2021 Edition” and *Vulnerability Disclosure Policy*. Accessed July 2026. Source for 90+30, the fourteen-day grace period, and the seven-plus-three in-the-wild track.

[24] Heelan, S. “How I Used o3 to Find CVE-2025-37899, a Remote Zero-Day in the Linux Kernel’s SMB Implementation.” May 2025.

[25] International Organization for Standardization. *ISO/IEC 9899: Programming Languages — C.* Cited for undefined behaviour (3.4.3), unsigned wraparound (6.2.5), the usual arithmetic conversions (6.3.1.8), and pointer arithmetic (6.5.6).

[26] International Organization for Standardization. *ISO/IEC 29147: Vulnerability Disclosure* and *ISO/IEC 30111: Vulnerability Handling Processes.*

[27] Kettle, J. “HTTP Desync Attacks: Request Smuggling Reborn.” PortSwigger Research, and *HTTP Desync Attacks: Smashing into the Cell Next Door*, Black Hat USA, 2019.

[28] Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. “Evaluating Fuzz Testing.” *ACM Conference on Computer and Communications Security (CCS)*, 2018. Source for the stack-hash de-duplication figures in §4.6 and §9.5, and for the trial-count survey.

[29] Chromium and libwebp projects. CVE-2023-4863, heap buffer overflow in **BuildHuffmanTable**; fixed in libwebp 1.3.2, 2023. See also the withdrawn duplicate identifier CVE-2023-5129.

[30] Miller, B. P., Fredriksen, L., and So, B. “An Empirical Study of the Reliability of UNIX Utilities.” *Communications of the ACM* 33(12), December 1990. Almost ninety utilities across seven versions of Unix; 25–33% crashed. The paper’s own introduction gives the line-noise-on-a-dial-up-line origin recounted in the footnote to §1.3.

[31] Netatalk project and NVD. CVE-2024-38440: off-by-one heap write in the DHX authentication path (**etc/uams/uams_dhx_pam.c**), reached through **FPLoginExt**;

fixed in Netatalk 3.2.1. Source for §3.3(v). A single **0** byte is written out of bounds, with no attacker choice over its value.

[32] Nethercote, N., and Seward, J. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation." *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[33] O'Callahan, R., Jones, C., Froyd, N., Huey, K., Noll, A., and Partush, N. "Engineering Record and Replay for Deployability." *USENIX Annual Technical Conference*, 2017.

[34] OpenSSL Project. *Vulnerabilities.* **openssl-library.org/news/vulnerabilities**. Accessed July 2026. The per-advisory credit lines are the source for the attribution counts in §9.6; the list is revised as advisories are issued, so any count taken from it carries its access date.

[35] Serebryany, K. "OSS-Fuzz — Google's Continuous Fuzzing Service for Open Source Software." *USENIX Security Symposium*, 2017; and the OSS-Fuzz project README, accessed July 2026, for the figures quoted in §1.3.

[36] Pencavel, J. "The Productivity of Working Hours." *The Economic Journal* 125(589), 2015, pp. 2052–2076. Re-analysis of the British Health of Munition Workers Committee observations. Output is proportional to hours below a threshold and rises at a decreasing rate above it. This is the measured evidence for the concavity of H in §6.4, and it is measured on production rather than on what anybody said about their own stamina.

[37] See, J. E., Howe, S. R., Warm, J. S., and Dember, W. N. "Meta-Analysis of the Sensitivity Decrement in Vigilance." *Psychological Bulletin* 117(2), 1995, pp. 230–249. The within-session form of the same effect: perceptual sensitivity declines with time on task across a large body of studies and task types.

[38] Van Dongen, H. P. A., Maislin, G., Mullington, J. M., and Dinges, D. F. "The Cumulative Cost of Additional Wakefulness: Dose-Response Effects on Neurobehavioral Functions and Sleep Physiology from Chronic Sleep Restriction and Total Sleep Deprivation." *Sleep* 26(2), 2003, pp. 117–126. Fourteen nights at 4 h or 6 h in bed produced cumulative, dose-dependent cognitive deficits. Source for the saturating recovery term in §6.4, and for the divergence between subjective sleepiness and objective impairment that Exercise 6-3 asks about: sleepiness ratings responded acutely and then rose only slightly, so subjects were largely unaware of deficits that continued to accumulate.

[39] Poeplau, S., and Francillon, A. “Symbolic Execution with SymCC: Don’t Interpret, Compile!” *USENIX Security Symposium*, 2020.

[40] Qualys Research Team. “Baron Samedit: Heap-Based Buffer Overflow in Sudo (CVE-2021-3156).” January 2021. Introduced in commit 8255ed69 (July 2011); affects 1.8.2–1.8.31p2 and 1.9.0–1.9.5p1.

[41] Qualys Research Team. “GHOST: **glibc gethostbyname** Buffer Overflow (CVE-2015-0235).” January 2015. Source for §3.3(v). The advisory gives the size expression in full and identifies the omitted **sizeof (*h_alias_ptr)** term; it also carries the Exim remote-code-execution proof, which is the reason a four-byte overflow of digits and dots was taken as seriously as it was.

[42] Qualys Research Team. “Sequoia: A Local Privilege Escalation Vulnerability in Linux’s Filesystem Layer (CVE-2021-33909).” July 2021. Source for §3.3(ii). Introduced by commit 058504ed in Linux 3.16 (July 2014). Qualys report full root on default Ubuntu 20.04, 20.10 and 21.04, Debian 11, and Fedora 34 Workstation; the exploit needs roughly 5 GB of memory and a million inodes, which is the detail that makes the point about what fuzzing was never going to reach.

[43] Google. *google/sanitizers* project wiki: *AddressSanitizerAlgorithm*, *AddressSanitizerFlags*, *AddressSanitizerIntraObjectOverflow*. Accessed July 2026. Source for the redzone and quarantine defaults in §2.5.

[44] Schloegel, M., Bars, N., Schiller, N., et al. “SoK: Prudent Evaluation Practices for Fuzzing.” *IEEE Symposium on Security and Privacy*, 2024.

[45] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. “AddressSanitizer: A Fast Address Sanity Checker.” *USENIX Annual Technical Conference*, 2012. The 73% slowdown and 3.37× memory figures are from this paper, measured on SPEC CPU2006.

[46] Seriot, N. “Parsing JSON is a Minefield.” 2016. “There were no two parsers that exhibited the same behaviour.”

[47] Shoshitaishvili, Y., Wang, R., Salls, C., et al. “SoK: (State of) The Art of War — Offensive Techniques in Binary Analysis.” *IEEE Symposium on Security and Privacy*, 2016.

[48] Spafford, E. H. “The Internet Worm Program: An Analysis.” Purdue University Technical Report CSD-TR-823, 1988.

[49] Stephens, N., Grosen, J., Salls, C., et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution.” *Network and Distributed System Security Symposium (NDSS)*, 2016.

[50] syzkaller project. *syzkaller: Kernel Fuzzer* documentation, including the **syzlang** description language. Accessed July 2026.

[51] Szekeres, L., Payer, M., Wei, T., and Song, D. “SoK: Eternal War in Memory.” *IEEE Symposium on Security and Privacy*, 2013.

[52] Vargha, A., and Delaney, H. D. “A Critique and Improvement of the CL Common Language Effect Size Statistic of McGraw and Wong.” *Journal of Educational and Behavioral Statistics* 25(2), 2000.

[53] Wang, X., Chen, H., Cheung, A., Jia, Z., Zeldovich, N., and Kaashoek, M. F. “Undefined Behavior: What Happened to My Code?” *Asia-Pacific Workshop on Systems (APSys)*, 2012.

[54] GNU Wget project; Red Hat Bugzilla 1505444. CVE-2017-13089: stack-based buffer overflow in **http.c:skip_short_body**, fixed in wget 1.19.2. 2017. Source for §3.3(iii). The chunk length is read with **strtol** and not checked for being negative; **fd_read** then narrows it to **int**. The sibling identifier CVE-2017-13090 is the heap-based case in the same parser.

[55] Zeller, A., and Hildebrandt, R. “Simplifying and Isolating Failure-Inducing Input.” *IEEE Transactions on Software Engineering* 28(2), February 2002. The *ddmin* algorithm and the definition of *n*-minimality.

Colophon

Composed in Times with Helvetica for display, after the manner of American technical-textbook publishing, and set by the author, slowly. The figures are drawn in the house style: line, label, and no shading that does not carry information.

That rule governed every illustration in the first two editions, and this one breaks it, so the break should be stated rather than discovered. The third edition admits photographs, set apart from the figures as *plates* and numbered separately, on one condition: a plate must carry an argument in its caption. Not a mood, not a texture, and not a picture of a person at a keyboard. Where a plate is only decorative it has been cut, and there are consequently five of them rather than the twenty a designer asked for. The distinction is the justification: a figure shows you data, a plate argues by analogy, and a reader is entitled to know which one is being done to them. The third edition abandons the Palatino of the first two, over the objection of exactly one reviewer, who was right about the italics and wrong about everything else.

The cover photograph is by Avi Theret, made in Wales and published on 3 October 2020, and is used under the Unsplash License, which permits commercial use and asks for no attribution. It is credited anyway, with the licence named and the date it was retrieved recorded, because a book that spends nine chapters telling you to check your sources should not put an unattributed asset on its own front. It has been converted to greyscale and cropped; nothing has been added to it or removed from it.

The interior plates are, in order: a canal lock, by Guus Seldenthuis; a false-colour infrared woodland, by Wolfgang Hasselmann; a repeating facade, by Pawel Czerwinski; a lit arch under stars, by Mike Newbry — all four under the Unsplash License, and all four reproduced in greyscale save the infrared, which is printed in colour because the colour is the instrument's output and desaturating it would have destroyed the only thing it was there to show. The fifth, a scale model in a supersonic wind tunnel, is a NASA photograph and is not subject to copyright; NASA asks to be acknowledged as the source, and is. It has been cropped to exclude the agency's insignia, which its media policy reserves. Earlier printings carried a public-domain silhouette in this position, and the sentence that accompanied it holds for the photograph too: it is here because a book about finding things in the dark should have an animal on it.

The corrected third printing fixes an inverted inequality in Exercise 4-2 (found by a

candidate, thanked in person; the error was mine and she was right in front of the cohort, which is the system working), replaces the crash-count figure in §4.6 with the sentence that should always have been there, and adds §9.6 at the request of the Directorate, who asked for it in writing — which in this institution constitutes both courage and a citation.

Errata to the author, campus mail, or to any duty reviewer, who will know where I am.

Out-of-universe notices. This is a work of fiction in its framing and a work of instruction in its content, and the reader is entitled to know exactly which is which.

Invented: Cyberpsychotic Press, the Institute for the Study of Adversarial Computation, the Assurance Compact, the Ninth Street campus, the Bench grades, the ARGUS review board, the syllabus codes, the cohort and incident references, the archive percentages in §3.2, and every named person (Jacob A., T. Vance, Y. Sorokin, D. Prather, R. Okoye, L. Wu) together with the works and theorems attributed to them. "Prather's theorem" was a pedagogical dramatisation of an ordinary path-induction argument, and both it and "the Okoye limit" have had their invented attributions withdrawn in this printing: §6.4's concavity claim now cites the working-hours and sleep-restriction literature instead of a colleague who does not exist. The characters remain; the citations are no longer theirs. No cats were consulted.

Real, and checkable: the tools (AFL++, libFuzzer, honggfuzz, Centipede, syzkaller, CodeQL, Semgrep, KLEE, angr, SymCC, Valgrind, **rr**, **llvm-cov**, libprotobuf-mutator), their described mechanisms including AddressSanitizer's 1:8 shadow mapping, redzones and quarantine and its published 73% slowdown and 3.37× memory costs; the vulnerabilities discussed by identifier (CVE-2021-3156 in sudo, found and published by the Qualys Research Team; CVE-2023-4863 in libwebp; CVE-2025-6965 in SQLITE) and their root causes as described by their finders and maintainers; the 1988 Morris worm, Miller's 1990 random-testing work, and Dowd, McDonald and Schuh's *The Art of Software Security Assessment* (2006); the memory-safety proportions reported by Microsoft, Chromium, and Android, including Android's decline from 76% in 2019 to 24% in 2024; the OSS-FUZZ figures (13,000+ vulnerabilities and 50,000+ bugs across roughly a thousand projects as of May 2025); Klees et al.'s 2018 fuzzing-evaluation guidelines and the 2024 systematisation showing how little they are followed; the documented over- and under-counting of stack-hash de-duplication; the disclosure policies of Google Project Zero (90+30, 14-day grace, 7-day in-the-wild), CERT/CC (45 days), and CISA; and the 2024–2026 record of language-model agents reporting real defects in widely deployed C code. Where this book paraphrases those sources it may have paraphrased them imperfectly; the originals are better than my summaries and are the reason the Notation and Conventions page tells you to read them.

Deliberately absent: working exploits, shellcode, gadget chains, allocator-grooming recipes, mitigation bypasses, and any target list. The line held throughout is the one stated in the Preface: this book goes as far as the minimal reproducer, the root cause, and an honest impact argument, and then stops, because that is what gets software fixed and it is also the whole of what can responsibly be taught in print.

Legal note, in earnest and not in character: the RANGE SAFETY boxes are the only part of this book that is not stylistic homage. Testing computer systems without authorisation is unlawful in most jurisdictions regardless of intent or outcome. Own the target, or hold specific written permission. Everything else here

is a technique; that one is a condition of practising at all.

On the model: the structure, apparatus, and voice of this volume are an affectionate pastiche of a syllabus text supplied by the reader, and the debt is acknowledged with gratitude.

Index

address-space layout randomisation, 5
AddressSanitizer, 5, 10
AFL, 5
AFL++, 29, 52
alignment, 9
allocator, 12
  free list, 12
  hardened, 12
  PartitionAlloc, 12
  Scudo, 12
  size class, 12, 72
Android, 7, 19, 38
angr, 38
attack surface, 23
  enumeration, 5, 25
authorisation, 7, 31, 46, 58, 68
automated discovery, 6, 43, 62

benchmark, 67
binary-only targets, 46
blast radius, 46
buffer overflow, 3, 10
  stack, 5, 11, 22
buffer underflow, 10

Centipede, 29
checked arithmetic builtins, 15
checksum, 36, 56, 74
CmpLog, 29, 48, 74
CodeQL, 25, 38
completeness argument, 17
concolic execution, 38
consequence class, 8, 71
container, 28, 68
control-flow enforcement, 53
corpus, 15, 28, 38, 47, 68
  seeds, 29, 47, 68
  synchronisation, 30
coverage, 5, 14, 28, 40, 48, 60
  edge, 28, 33, 74
  report, 30, 43, 48, 68
crash de-duplication, 31, 61
CVE
  2015-0235 (GHOST), 21
  2015-1538 (Stagefright), 19
  2016-2347 (Lhasa), 20
  2017-13089 (wget), 20
  2021-33909 (Sequoia), 19
  2023-4863 (libwebp), 25
  2024-38440 (Netatalk), 21
  2025-6965 (SQLite), 8

dangling pointer, 10
data race, 15
dataflow analysis, 24, 37, 76
debugger, 14, 54, 65, 79
decoder, 31, 34, 43, 60, 68
delta debugging, 53
  1-minimal, 54, 79
deserialisation, 23
dictionary (fuzzing), 23, 28, 48, 68
disclosure
  bounty programme, 57, 79
  coordinated, 57
  CVE identifier, 57

embargo, 57
grace period, 57, 79
numbering authority, 57
Project Zero policy, 57
timeline, 57, 69
double free, 10
Driller, 38
dynamic binary instrumentation, 49

effect size, 32, 75
emulation, 49
encoder (fuzzing the), 31, 43, 48, 68
exclusions section, 41, 48
Exim, 21
exploitability, 56, 61, 65, 70

fidelity, 35, 60
file descriptor, 50
firmware, 49
FRIDA, 49
fuzzing
coverage-guided, 32
greybox, 62
hybrid, 38
persistent mode, 36, 49
structure-aware, 31, 34, 41, 49, 74

GHOST, 21
Ghostscript, 24, 68
GitHub Security Lab, 24
guard, 18, 63

harness, 19, 23, 28, 34, 41, 47, 66, 68
contract, 33, 50, 57, 68
over-reach, 47
rot, 48
honggfuzz, 29
Huffman table, 23

input-to-state substitution, 32, 38
integer overflow, 10
multiplication, 73
subtraction underflow, 20
truncation, 73
unsigned wraparound, 15, 73
intra-object overflow, 10, 21
invariant, 26, 31, 37, 48, 67
IPC, 23
issue tracker, 25, 41, 57

JSON, 37

KLEE, 38

LAF-Intel, 29
Lhasa, 20
libFuzzer, 29, 36, 53, 66
libwebp, 23

magic number, 27
maintainer, 31, 53
Mann–Whitney U test, 32, 75
memory safety, 6
MemorySanitizer, 13, 36
Miller, B. P., 5
minimisation, 36, 62, 79
mitigation, 11, 56
Morris worm, 5

namespace, 31
Netatalk, 21
non-executable pages, 5

object lifetime, 9
off-by-one, 10, 21
oracle, 6, 9, 30, 33, 48, 61, 66, 67
assertions, 31, 48, 74
differential, 39, 48, 68

round-trip, 37, 48
OSS-Fuzz, 31, 51
out-of-bounds read, 10, 22

parser differential, 7
path explosion, 38
Pencavel, J., 44
plateau, 27, 68
pointer provenance, 9, 18, 42, 48, 75
primitive (impact description), 10, 53, 70
proof of concept, 53
protocol buffer, 29
provenance table, 48

QEMU, 49
Qualys, 19
quarantine, 13, 68

reachability, 1, 23, 36, 56, 72
reconnaissance, 40, 47, 68
record-and-replay debugging, 54
Redqueen, 29
redzone, 13, 72
regression, 55, 69
reproducer, 47, 53, 80
request smuggling, 37
return address, 11
reverse engineering, 49, 78
root cause, 7, 69
rotation policy, 30, 42, 49
round-trip oracle, 37
Rust, 6

sanitiser, 12, 25, 31, 47, 55, 69
AddressSanitizer, 5, 10
KASAN, 34, 67
LeakSanitizer, 36, 67
MemorySanitizer, 13, 36
UndefinedBehaviorSanitizer, 11, 19, 30, 47, 67
scope document, 7, 15, 78
scoped assurance, 41
seccomp, 31
Semgrep, 25
Sequoia, 19
severity, 12, 53, 61, 68
shadow memory, 13, 49, 72
signedness confusion, 20
snapshot and restore, 51, 63, 79
software reliability growth models, 61
spatial safety, 10
SPEC CPU2006, 7, 13
species discovery, 62
SQLite, 8
stack canary, 11, 53
stack frame, 11
Stagefright, 19
standing review, 48
statement of work, 41
substitution rule, 32, 38, 43
sudo, 22
symbolic execution, 38
SymCC, 38

taint analysis, 25
temporal safety, 10
ThreadSanitizer, 14
threat model, 5, 41, 68
throughput, 15, 30, 40, 49
triage, 24, 38, 53, 62, 74
trial count, 32, 39, 75
trust boundary, 1, 17, 71
type confusion, 10

undefined behaviour, 9

optimiser licence, 9
signed overflow, 10
UndefinedBehaviorSanitizer, 11, 72
Unicorn, 49
uninitialised read, 8, 14, 36
use-after-free, 3, 10, 54, 65
use-after-return, 10
use-after-scope, 10

Valgrind, 7, 14, 49, 67
origin tracking, 67
Van Dongen, H. P. A., 42, 77
variant analysis, 6, 41, 48, 68
virtual machine, 35, 50, 75

watchpoint, 54
wget, 20

Zeller, A., 53